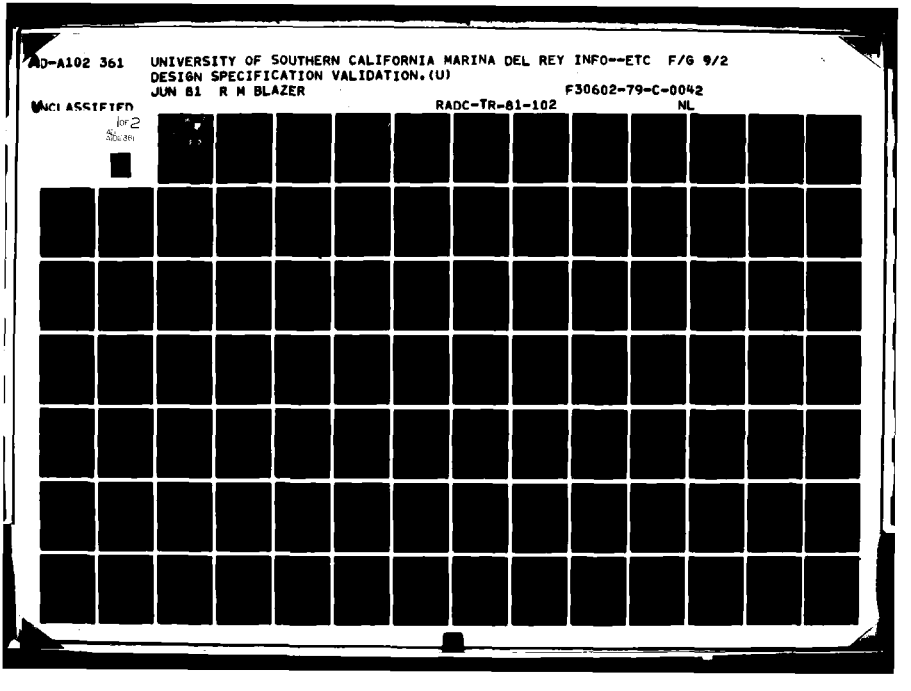
AD-A102 361 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
DESIGN SPECIFICATION VALIDATION.(U)
JUN 81 R M BLAZER F30602-79-C-0042
UNCLASSIFIED RADC-TR-81-102 NL



LEVEL II

12

RADC-TR-81-102
Final Technical Report
June 1981

DESIGN SPECIFICATION VALIDATION

University of Southern California

Robert M. Blazer

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-102 has been reviewed and is approved for publication.

APPROVED: Rocco F. Iuorno

ROCCO F. IUORNO
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-81-102	2. GOVT ACCESSION NO.: AD-A102 361
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): DESIGN SPECIFICATION VALIDATION	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. 29 Sep 79 — 29 Dec 80
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Robert M. Blazer	8. CONTRACT OR GRANT NUMBER(s): F30602-79-C-0042
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Southern California, Information Sciences Institute, Marina Del Rey CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55811817
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIE), Griffiss AFB NY 13441	12. REPORT DATE: June 1981 13. NUMBER OF PAGES: 178
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software design, Software specifications, Software requirements, Symbolic execution, Requirements language, Software design language, Specification language, Software testing, Validation.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report documents the research performed by the University of Southern California, Information Sciences Institute concerning the feasibility of validating formal software specification. The one year effort attempted to show that with a suitable formal software specification language, specifications could be validated in the same manner as computer designs are currently tested.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Table of Contents

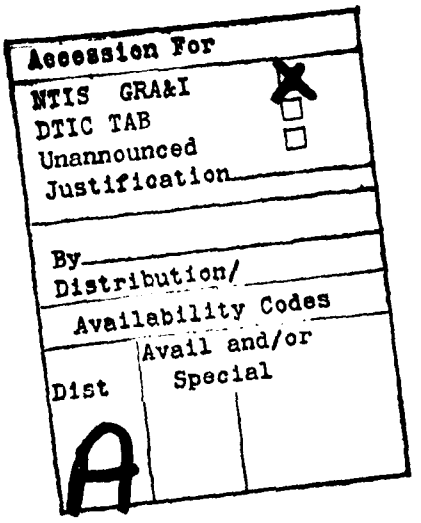
1. SUMMARY 1-1

2. GIST SPECIFIER'S MANUAL 2-1

2.1 INTRODUCTION 2-1
2.2 TYPES 2-2
2.3 RELATIONS AND ATTRIBUTES 2-4
2.3.1 Attributes 2-5
2.3.2 Attributes and Subtypes 2-7
2.4 CONSTRAINTS 2-7
2.4.1 Use of Constraints 2-10
2.5 DERIVATION RULES 2-11
2.6 OBJECT EXPRESSIONS 2-12
2.6.1 Literals 2-12
2.6.2 Variables 2-12
2.6.3 Attribute Reference 2-13
2.6.4 Descriptive Reference 2-13
2.6.5 Arithmetic Expressions 2-14
2.7 PREDICATES 2-14
2.7.1 Propositions 2-14
2.7.2 Quantified Predicates 2-16
2.7.3 Predicate Composition 2-17
2.8 ACTION DECLARATIONS 2-17
2.8.1 Primitive Statements 2-20
2.8.1.1 Object Creation 2-20
2.8.1.2 Object Destruction 2-21
2.8.1.3 Adding Relationships and Classifications 2-21
2.8.1.4 Removing Relationships and Classifications 2-22
2.8.1.5 Updating Relationships 2-22
2.8.2 Action Invocation 2-23
2.8.3 Compound Statements 2-23
2.8.3.1 Conditional Behavior 2-23
2.8.3.2 Sequential Behavior 2-24
2.8.3.3 Point Invariants 2-24
2.8.3.4 Alternative Behavior 2-25
2.8.3.5 Preferential Behavior 2-25
2.8.3.6 Non-primitive Granularity 2-26
2.8.3.7 Iterative Behavior 2-26
2.9 ADVANCED TOPICS 2-28
2.9.1 Sets and Sequences 2-28
2.9.2 Multiple Lines of Control 2-28
2.9.3 Temporal Reference 2-28
2.9.4 Boundaries 2-29
2.9.5 Orderings 2-29
2.9.6 Meta Concepts 2-29

3. GENERAL APPROACH TO WRITING GIST SPECIFICATIONS 3-1

3.1 IDENTIFYING TYPES AND SUBTYPES 3-1



3.2 IDENTIFYING INDIVIDUALS 3-2
3.3 IDENTIFYING RELATIONS 3-2
3.4 IDENTIFYING DERIVED CONCEPTS 3-2
3.5 IDENTIFYING STATIC CONSTRAINTS 3-3
3.6 IDENTIFYING ACTIONS 3-3
3.7 IDENTIFYING DYNAMIC CONSTRAINTS 3-4
3.8 IDENTIFYING ACTIVE PARTICIPANTS 3-4
3.9 PROGRESSING IN COMPETENCE 3-4

4. CONSTRUCTING GIST SPECIFICATIONS 4-1

4.1 SOURCE-DATA MAINTENANCE EXAMPLE 4-1
4.1.1 Suppressed details 4-2
4.1.2 Plan for Constructing Source-Data Maintenance Specification 4-4
4.1.2.1 Modeling objects and relationships of domain 4-4
4.1.2.1.1 Notation for types and binary relations 4-5
4.1.2.1.2 Type and attribute definitions for the domain 4-7
4.1.2.2 Static constraints 4-8
4.1.2.2.1 A "static" constraint 4-8
4.1.2.2.2 The use of derivation 4-9
4.1.2.3 Modeling user commands 4-10
4.1.2.4 Modeling change 4-12
4.1.2.4.1 Object boundaries 4-12
4.1.2.4.2 The top-level action to support user-commands 4-13
4.1.2.4.3 A simple action 4-14
4.1.2.4.4 Adjusting process "granularity" 4-15
4.1.2.4.5 Non-determinism and constraints 4-15
4.1.2.4.6 Shifting characters on a line 4-16
4.1.2.4.7 The use of historical reference 4-17
4.1.2.4.8 Historical reference and inserting lines into units 4-18
4.1.2.4.9 Historical reference and backup/restore commands 4-18
4.1.2.4.10 Historical reference and desired behavior 4-19
4.1.2.5 Dynamic constraints 4-21
4.1.2.5.1 A "dynamic" constraint 4-21
4.1.3 Review of specification 4-21
4.1.4 Implications for Gist 4-22
4.1.5 Implications for the source-data maintenance task 4-22
4.1.6 Difficulties of constructing the specification 4-22
4.1.7 Difficulties of understanding specification 4-23
4.1.8 Use of specification 4-23
4.2 HOST-IMP SPECIFICATION 4-24
4.2.1 Overall organization of specification 4-24
4.2.2 General message-passing features 4-24
4.2.3 Specialization to host-imp world 4-25
4.2.4 Implications for Gist 4-27
4.3 TEXT FORMATTER SPECIFICATION 4-27
4.3.1 Source of Problem 4-27
4.3.2 Formatter activities 4-29
4.3.3 Interface 4-30
4.3.4 Organization of specification 4-31
4.3.5 Implications for Gist 4-32

4.3.5.1 New Gist usage 4-32
4.3.5.2 Exposed weaknesses of Gist and dissatisfaction with specification 4-32
4.3.5.3 Advantages accrued from the use of Gist 4-33
4.3.6 Extensions to formatter 4-33
4.3.6.1 Separating justification and filling 4-34
4.3.6.2 Filtering output pages 4-34
4.3.6.3 Forcing text to appear on a single page 4-34
4.3.6.4 Extra space after sentence 4-34
4.3.6.5 Hyphenation during filling 4-35
4.3.6.6 Conditionals 4-35

5. GIST INITIAL OPERATING CAPABILITY DESCRIPTION 5-1

5.1 INTRODUCTION 5-1
5.1.1 Transliteration 5-3
5.2 Editor, PrettyPrinter, and File I/O 5-3
5.2.1 Commands which Alter Specification Text 5-4
5.2.2 Commands which Change the Editor's Focus 5-5
5.2.3 PrettyPrinting 5-6
5.2.4 Commands for Saving and Restoring Specifications 5-6
5.3 Specification Testing 5-7
5.3.1 The Gist Declaration Compiler 5-7
5.3.2 The Gist Evaluator 5-7
5.3.3 Literal and Created Objects 5-9
5.3.4 Input/Output Facilities 5-9
5.4 Debugging/Testing Aids 5-10
5.4.1 Modifying a Specification 5-11
5.5 IOC Coverage 5-11

6. IOC TRACES 6-1

6.1 IOC TRACE--PSL EXAMPLE 6-1
6.2 IOC TRACE -- CAMELOT EXAMPLE 6-12

7. DESIGN FOR A GIST SPECIFICATION VALIDATION FACILITY 7-1

7.1 OBJECTIVE 7-1
7.2 BACKGROUND 7-1
7.3 APPROACH 7-4
7.4 PLAN 7-5

Appendix I. GIST GRAMMAR 7-7

Appendix II. GIST SPECIFICATIONS I-1

II.1 SOURCE DATA MAINTENANCE SPECIFICATION II-1
II.1.1 *** Type definitions for objects of domain *** II-1
II.1.2 *** Static constraints on the world *** II-2
II.1.3 *** User commands *** II-3
II.1.4 *** Dynamic constraints on the world *** II-5
II.1.5 *** Interface with user *** II-6
II.1.6 *** Actions to perform commands *** II-6
II.1.6.1 *** SOURCE-DATA-MAINTENANCE *** II-6
II.1.6.2 *** UPDATE *** II-6
II.1.6.3 *** ADD *** II-7

II.1.6.4 *** PURGE *** II-7
II.1.6.5 *** REPLACE *** II-8
II.1.6.6 *** COPY-UNIT *** II-8
II.1.6.7 *** CHANGE *** II-8
II.1.6.8 *** INSERT *** II-9
II.1.6.9 *** DELETE *** II-10
II.1.6.10 *** SHIFT *** II-11
II.1.6.11 *** MODIFY *** II-11
II.1.6.12 *** COPY-LINES *** II-13
II.1.6.13 *** REPLACE-LINE *** II-13
II.1.6.14 *** COPY-FILE *** II-13
II.1.6.15 *** BACKUP and RESTORE *** II-13
II.1.7 *** Restoration following temporary changes *** II-15
II.1.8 *** Maintenance of information about units *** II-16
II.2 HOST-IMP SPECIFICATION II-17
II.3 FORMATTER SPECIFICATION II-23
II.3.1 Definitions of top level types II-24
II.3.2 FORMAT (action) II-26
II.3.2.1 CREATE←LINELETS←FROM←MIXLETS (action) II-27
II.3.2.2 PAGINATION (action) II-28
II.3.2.2.1 PAGE←PARTITION (relation) II-29
II.3.2.2.2 CREATE←PAGE←IMAGE (action) II-30
II.3.2.2.2.1 PAGE←NUMBER (relation) II-31
II.3.2.2.2.1.1 PRECEDING←PAGE←NUMBER (relation) II-31
II.3.2.2.2.2 STRIP←LEADING←AND←TRAILING←BLANKS (relation) II-32
II.3.2.2.2.3 LINES←FROM←LINELETS (relation) II-33
II.3.2.2.2.4 CREATE←TITLE←LINES (action) II-34

II.3.2.3 PARAGRAPHING (demon) II-35
II.3.2.3.1 CREATE←JUSITIFIED←INFO←LINES←FOR←PARAGRAPH (action) II-36
II.3.2.3.1.1 CREATE←JUSTIFIED←INFO←LINE←FOR←WORDS (action) II-37
II.3.2.3.1.1.1 CREATE←JUSTIFIED←LINE←FOR←WORDS (action) II-38
II.3.2.3.1.1.1.1 ARBITRARILY←PAD←WORDS←WITH←BLANKS II-39
II.3.2.3.1.1.1.1.1 PAD←INTERIOR←WITH←BLANKS (action) II-40
II.3.2.3.1.2 CREATE←UNJUSTIFIED←INFO←LINE←FOR←WORDS (action) II-41
II.3.2.3.1.2.1 ARBITRARILY←PAD←WORDS←WITH←BLANKS (action) II-39
II.3.2.3.1.2.1.1 PAD←INTERIOR←WITH←BLANKS (action) II-40

II.3.2.4 SPLIT←OVERLENGTH←LINES (demon) II-42
II.3.2.4.1 CREATE←SPLIT←LINES←FOR←OVERLENGTH←LINE (action) II-43
II.3.2.4.1.1 SUBSEQUENTIZE←INFO (action) II-44

II.3.2.5 LEFT←MARGIN←PADDING (demon) II-45
II.3.2.5.1 EXTEND←BLANK←SEQUENCE (action) II-46

II.3.2.6 CENTERING (demon) II-47

II.3.2.7 INTER←LINE←PADDING (demon) II-48
II.3.2.7.1 EXTEND←PADDING←LINE←SEQUENCE (action) II-49
II.3.2.7.1.1 SUBSEQUENTIZE←INFO (action) II-44

REFERENCES

List of Figures

Figure 2-1: Action Declaration in Gist 2-18
Figure 4-1: Structure of source-data maintenance domain 4-2
Figure 4-2: User commands supported by the system 4-3
Figure 4-3: Modeling of objects and relations in Gist 4-5
Figure 5-1: Gist IOC configuration 5-2

List of Tables

Table 5-1: IOC symbol transliteration 5-3

1. SUMMARY

This report documents our findings concerning the feasibility of validating formal software specifications. This one-year effort attempted to show that with a suitable formal software specification language, specifications could be validated much as implementations are currently validated, through application of testing methodology.

This approach necessitated an "executable" formal specification language so that test cases could be "run" on the specification itself. Such a language was already under development within USC/Information Sciences Institute. Our first task was to document this language (see Chapter 2) and our approach to using it to write formal specifications (see Chapter 3). We then applied it to three real, moderate-sized systems (see Chapter 4) to ensure that it was suitable for military applications. These experiments were highly successful in that we were able to formally specify the functional behavior of these three systems.

The next major task of this effort was to demonstrate that specifications in this formal language could be validated by running them on test data. Towards this end, we built a prototype interactive testing facility called the Initial Operating Capability (IOC). It consists of an evaluator capable of executing specifications expressed in a subset of the formal specification language, and an executive for entering, editing, and displaying specifications, for initializing the state within which one is evaluated, for displaying states, and for tracing and breakpointing the evaluation interactively in a DDT-like manner. The IOC is described in Chapter 5 and examples of its use are given in Chapter 6.

The final task in this effort was to design a capability for testing a specification on a whole class of cases simultaneously, rather than one at a time, through symbolic execution. This design is documented in Chapter 7 of this report. It is based on extending the evaluator to handle symbolic data as well as concrete data, so that as evaluation proceeds, predicates describing incompletely defined data (i.e., the symbolic data) are automatically constructed dynamically, and wherever incompleteness of the data prevents determination of which control path to follow, all possible control paths are explored, each conditioned by the predicate necessary to select it.

Thus, there is tradeoff between breadth of test cases covered and complexity of the symbolic evaluation. For this reason, the design allows the person validating the specification to interactively determine which data should be concrete versus symbolic, and hence, dynamically define the class of test cases being explored.

The following observations and conclusions emerged from this study:

1. While functionally complete, the formal specifications omitted some aspects of the actual systems.

   These omissions were *intentional*. We have tried to prevent implementation details from intruding into the formal specification. For that reason data representations are not part of our specification language. This prevents us from describing some (possibly required) aspects of the specified system, most noticeably their I/O interfaces. We believe these (required) representatives should be documented as part of the chosen (required) implementation of the system, rather than as part of its specification.

2. Existing specifications incorporate implementation details.

   In order to get a good system description for which we could build a formal specification, we had to de-compile the existing description so that it only described the functionality desired, rather than an implementation of it. As current or former programmers, we are all overly sensitized to efficiency issues and tend to describe a system in terms of a feasible (or possibly preferred) implementation. Representation issues are one example; algorithm choices are another.

3. Formal specifications are unreadable.

   This is unfortunately true for our language as well as all others. The reason is that the mechanisms used in informal communications to aid understanding--such as overview, summaries, examples, elborations, alternative viewpoints, role descriptions (why something is important)--are totally missing. Basically, no roadmap is provided to help someone read the specification. Until these issues are addressed, formal specifications will only be understandable when accompanied by an informal natural language description.

4. The formal specification language is not interpretable.

   The reason is that the formal specification language is highly context dependent and this context is affected by many remote statements such as type declarations, constraints, demons, inference rules, action definitions, and individual uses of historical reference. All these statements must be located, collected, and "translated" before interpretation can begin. Thus, the evaluator consists of a compiler which performs these tasks and an interpreter which uses these compiled forms.

5. The evaluator must be interactive.

   The formal specification language is highly non-deterministic and the non-determinism interacts with the specified constraints so that only those choices which don't violate constraints either now or in the future are selected. Operationally, the evaluator can only discover the appropriate choices by trying them and rejecting those that violate constraints anywhere during the evaluation. This leads to a classical backtracking search. To control the size and depth of this search space during evaluation, the person using the IOC can supply an appropriate choice rather than have the evaluator search for one.

2 GIST SPECIFIER'S MANUAL

2.1 INTRODUCTION

A Gist specification is a formal description of *valid behaviors* of a system. This description is composed of three parts:

- A specification of object types and relations between these types. This determines a set of *possible states*. A possible state consists of a collection of objects, each of which is classified as an instance of one or more of the specified types. The objects in a possible state may be related by the specified relations.
- A specification of *actions* and *demons*, which define transitions between possible states.
- A specification of *constraints* on states and state transitions. The *valid behaviors* of a system are those defined transition sequences that do not violate any constraints.

This document provides a tutorial introduction to Gist. The concepts and syntax of Gist are presented in the context of describing the behavior of an imaginary system of ships moving cargos among various ports.

A second goal of this introduction is to provide some insight into our conception of "good taste" in writing Gist specifications. In Gist, as in programming languages, there are often several ways to specify the same behavior (functionality). In programming languages, one must make tradeoffs among understandability of code, modifiability, and efficiency. In Gist, efficiency is not an issue; there is no sense in which two specifications of the same behavior differ in efficiency. The main goal of the specifier is to maximize his confidence that he has specified the intended behavior. But important secondary goals are to make a specification understandable to people other than its creator, and to make it maintainable. The rules of thumb and good practices described are aimed at these latter goals.

As with any language, the use of common conventions can itself improve communication between people. But the specifier should always remember that his overriding concern is to be confident that he is specifying the behavior intended.

While a primary purpose of writing a specification is to use it in the process of implementing computer software, there are no software concepts within Gist. In particular, all behavior is specified without the use of the concepts of input/output or the information representations available on any particular machine or in any particular programming language(s).

2.2 TYPES

The first task of a specifier is to decide on a collection of object types that will capture the important objects, or values, that the process manipulates. Since there is usually some verbal description of the process available, a good rule of thumb is to consider common nouns (particularly concrete nouns) as the names for object types.

Following this rule, even the cursory description of our example domain given earlier suggests the use of **ship**, **cargo**, and **port** as object types.

The names to be used for object types in Gist must appear in type declarations. The simplest form of a type declaration gives nothing but the name:

```
type ship;
type port;
type cargo;
```

Gist has only a few predefined types. Among these are number,[1] integer, natnum (natural numbers), and character. These types may be used in a specification without declaration. Sometimes one of these types is appropriate, but the specification would be more natural if a different name were used. This can be done by defining a new type name in terms of an existing one:

```
type tonnage definition natnum
```

The previous line defines the type name "tonnage" as isomorphic to the type "natnum".

The only reason to use the predefined types, however, is to use some capability that is already defined for them. The numerical types, for example, provide access to arithmetic operations, comparisons, and the ability to include numerical constants in the specification.

It is a bad practice to specify types to be synonyms for numbers because one anticipates the use of numbers as a *representation* in some implementation.

An enormous gain in expressiveness is obtained by specifying *supertype* relationships among types. If all objects of one class also belong to a second class, the larger class should be declared to be a supertype of the smaller class:

---

[1] Number corresponds to the REALs. It should not be thought of as specifying "floating point" representation on a computer.

```
type cargo unique supertype of
  < grain;
    fuel
  >
```

This declaration states that every instance of cargo is an instance of either grain or fuel, but not both.

In other situations, the named subtypes may not exhaust the supertype:

```
type ship optional supertype of
  < oiltanker;
    cruiseship
  >
```

The word *optional* indicates that a ship may be either an oiltanker or a cruiseship (although not both), but may be neither.[2]

Although the verbal names used for types often give no clue to supertype connections, as with the names "ship" and "oiltanker", certain patterns should be considered. Noun pairs, such as "cruise ship", are frequently used to name subtypes of the second noun, as are compound nouns. Adjectives frequently are used with nouns in the same way, as in "*military* vessel".

In order to fully describe a process, it is sometimes necessary to refer to individual objects (values) as well as to entire classes. The names of individuals may be freely chosen by the specifier, but the type to which each belongs must be declared. This can be done as part of the type declaration:

```
type port includes {SantaBarbara, Seattle};
type grain definition {Corn, Wheat};
type fuel definition {Oil, NaturalGas}
```

The declaration can either enumerate some of the instances of the types (includes) or may enumerate them all (definition).

Linguistic clues that indicate the need for declaring individual instances of types include the use of proper nouns, mass nouns, and collections of adjectives (like color names) used to modify the same type names.

The primary motive for declaring various types of objects in a specification is that the instances of a type are perceived to share characteristics not common to instances of other types. These include:

---

[2] Gist also permits declarations of "overlapping" subtypes, and types with multiple supertypes.

- relationships that hold between the objects
- actions that may be performed on the objects
- constraints on the objects

The remainder of the declaration portion of Gist is concerned with specifying these regularities.

2.3 RELATIONS AND ATTRIBUTES

In describing a process in English, we use specific words and syntax to describe the ways objects are related to one another. In "ship bound for SantaBarbara" the phrase "bound for" indicates a particular relation that can hold between ships and ports--namely, a ship can be scheduled to go to a particular port. "A ship based in SantaBarbara" uses a different phrase, "based in" to talk about a different relation between the same two types. "A ship containing 50 tons of wheat" indicates yet another relation in this domain--ships may contain specific quantities of specific cargos.

In Gist, these various kinds of associations that may relate the individual instances of the types are called *relations*. A relation is declared by giving it a name and by naming the object types it relates:

relation CONTAINS (SHIP, CARGO, TONNAGE) (2.1)

This does not specify which instances of ship, cargo, and volume are related or when such relationships arise. But it does say that the relation name **CONTAINS** will *not* be used to relate objects of any types other than the three named.

If in a given state the ship *USS-Prairie* contained *50* tons of *Wheat*, we would say in Gist terminology, that there was a relationship in **CONTAINS** having *USS-Prairie* filling the SHIP role, *Wheat* filling the CARGO role, and *50* filling the TONNAGE role.

Technically, each role of a relation has both a name and a type. In many cases, the specifier will feel comfortable using the same name for the role as was used for its type. This is done by using only the type name in the declaration, as in (2.1) above. In other cases, there is mnemonic value in choosing a name for the role that is more expressive than the type name. Finally, there are constructs in Gist that use the role name explicitly. For relations that have more than one role of the same type, these constructs are useable only if those roles have been given distinct names. Since it is a bad idea to try to anticipate which roles one may need to reference by name in a specification, one should always choose distinct names for two roles of the same type in the same relation.

There are two ways to indicate a name for a role of a relation when it is to differ from the type name.

One is to choose as the role name a composite name of the form "typename.distinguisher"--e.g., "COUNTRY.FROM". In this case, the composite name is the role name while its prefix (in this case, "country") is the role type. Alternatively, one can choose an arbitrary name for the role and specify its type explicitly, separating the two names by a "|" in the declaration. Suppose we wanted to have a relation that indicated what goods were being traded among nations. We could do this with a relation declared by:

relation TRADE (COUNTRY.FROM, GOODS | cargo, COUNTRY.TO)

making TRADE a ternary relation having roles named COUNTRY.FROM, GOODS, and COUNTRY.TO, whose types are country, cargo, and country, respectively.

In general, one can imagine countless relationships in any moderately rich process domain. It is not possible to determine in advance precisely which ones will be sufficient, or convenient, to use for the entire specification. A good rule of thumb, however, is to start with those mentioned in the verbal process description. Linguistic clues to relationships include:

- the use of "situational" verbs like "contain" or "own",
- the use of passive forms like "be connected to" or "be bound for",
- noun phrases modified by prepositional phrases, like "the berth of a ship" or "the salary of an officer",
- possessive forms, like "the ship's country of registry", or "the officer's salary".

2.3.1 Attributes

It is usually the case that a large percentage of the useful relations in a specification will be *binary* relations. These can be declared and used like ordinary N-ary relations, but may also be declared and used in a special *attribute* notation in Gist. The use of attribute notation makes the declaration of the relationship, its use, and the specification of constraints on it more concise and, arguably, easier to understand.

Attributes are declared as part of type declarations. For example, to declare the binary relationship between ships and their country of registry, the attribute is named in the declaration of one of the types:

type ship (REGISTRY | country);

This declaration indicates that ships and countries are related by an attribute named REGISTRY. Elsewhere in Gist, ": REGISTRY" is used to designate the mapping for ships to countries, and ":: REGISTRY" to designate the inverse mapping from countries to ships.

Many attributes may be declared in a single type declaration. The declaration:

type **ship** (REGISTRY | **country**, CAPACITY | **tonnage**, DEADWEIGHT | **tonnage**) (2.2)

specifies the existence of three mappings, whose names are intended to convey the meaning of the mappings.

The declaration of relationships, including attributes, should be made for the most general applicable types. It is then "inherited" by the subtypes of those types. For example, having declared the attribute "REGISTRY" between ships and countries, there is no need to duplicate the declaration for subtypes of ship, such as oiltanker, or for subtypes of country.

Having decided to use an attribute declaration as opposed to a binary relation, a specifier still must decide in which of the two relevant type declarations to include the declaration of the attribute.[3,4] It is impossible to do more than give guidelines on this matter, since there is no formal distinction in the declaration content. As is frequently the case with guidelines they may suggest conflicting organizations.

- Linguistic guideline. Choose an attribute name that makes the phrase:

  ⟨attribute name⟩ *of/in/for/* ⟨type⟩$_1$ *must be* ⟨type⟩$_2$.

  sound like a sensible description of the relationship. Declare the attribute within the declaration of type$_1$. Each attribute in (2.2) above would be suggested by this rule, since:

  registry of ship must be country
  capacity of ship must be tonnage
  deadweight of ship must be tonnage

  sound right.

- Functionality guideline. If the mapping is many-to-one--each object of type$_1$ maps to exactly one object of type$_2$, but an object of type$_2$ may map to zero, one, or more objects of type$_1$--then declare the attribute with the declaration of type$_1$. All three attributes in (2.2) suggest this.[5]

- Structural guideline. Some relationships have a "part-whole" feeling to the specifier. Sometimes this has a strong physical basis in the real world, as with the slips of a pier. Other times the feeling is more one of a logical dependency specific to the process under consideration, as with the dependents of an employee. Try to declare such attributes as part of the declaration of the "whole" type.

---

[3] It could be redundantly stated in both places, but this would not be a good idea in general, since any change to the declaration would require two changes in the specification to maintain consistency.

[4] Of course, if the two types are the same, there is no problem.

[5] Certain default constraints on the mappings, described later, are designed to mesh well with this guideline.

It is possible, by means described in Section 2.4, to specify constraints on attribute mappings along with the attribute declaration itself. Sometimes the restriction of the mapping that involves a named subtype of one of the two types is more highly constrained than the mapping as a whole. For instance, crewmember's salaries may range from $10,000 to $40,000, but officer's salaries may have to be above $25,000. When this situation arises, more effective use can be made of Gist's constructs if the attribute is declared with the type whose subtype is further constrained--crewmember, in this example.

2.3.2 Attributes and Subtypes

Whenever a proposed attribute relationship involves a type whose instances are totally enumerated by literals in the specification, the specifier should consider the possibility that a more natural and useful specification could be obtained by using the supertype concept. In other words, the attribute relationship between $type_A$ and enumerated $type_B$ can be replaced by having a named subtype of $type_A$ for each instance of $type_B$, and eliminating $type_B$ entirely. A particularly common situation to be aware of is the use of "flag" types whose instances are "yes" and "no", or "true" and "false", or 0 and 1. It is almost always preferable to provide a name for each case and make the encompassing type a "unique supertype of" these named types, or to name one of the cases (typically, the "positive" one) and make the encompassing type an "optional supertype of" that named type.

For example, it would be preferable to specify the fact that some ships are government-owned as:

```
type ship(...) optional supertype of
  <governmentship (...)>;
```

rather than:

```
type ship(...,GOVERNMENTOWNED | {Yes,No})
```

2.4 CONSTRAINTS

The declaration of types and relationships lays out a wide variety of possible process states--that is, states containing collections of instances of those types and relationships between objects of appropriate types. However, it is nearly always obvious that many of these states may never really arise. Sometimes this is because of physical constraints in the process domain--e.g., a ship cannot simultaneously occupy multiple slips, nor can multiple ships simultaneously occupy one slip. In other cases, it is not physical reality, but the desired process which restricts the potential states. For example, there may be a "rule" which states that no ship should simultaneously carry fuel and grain. In either case, Gist provides declarations for specifying such constraints on possible states.

The most general construct permits the specifier to state that a given condition must either:

- hold in every process state (a *requirement*)
- hold in no process state (a *prohibition*).

The constraining condition is specified by a Gist predicate. Predicates are discussed in more detail in Section 2.7, but basically follow conventional predicate calculus notation. For example,

```
always prohibited ∃ ship,fuel,grain || CONTAINS(ship, fuel, $) ∧
                                       CONTAINS(ship, grain, $)
```

would state that no ship may ever be used to *simultaneously* transport fuel *and* grain.

```
always required ∀ oiltanker || ∃ officer || officer : ASSIGNMENT = oiltanker ∧
                                            officer : SENIORITY > 10
```

would require that officer assignments ensure that every oiltanker have at least one officer with over 10 years experience.

Although it is possible to describe all Gist's constraints in this way, experience shows that many constraints can be naturally expressed as constraints "on" particular relations or types. For such constraints Gist provides a means of expressing the constraint as part of the type or relation declaration.

With respect to a given relation, there are two classes of constraints which may be specified. The first concerns constraints on individual relationships in the relation. For instance, to prohibit trade of any product between two countries which are both producers of that product, one could place a constraint on the TRADE relation in terms of a PRODUCES relation:

```
relation PRODUCES (COUNTRY, GOODS | cargo);
relation TRADE (COUNTRY.FROM, GOODS | cargo, COUNTRY.TO)
 where always prohibit PRODUCES(country.from, goods) ∧
                       PRODUCES(country.to, goods)
 end
```

The second form of relation-specific constraint restricts the *collection* of relationships which may co-exist in a given relation. For any partitioning of the roles of a relation into two non-empty partitions, one may view the relation as a mapping from one partition to the other. It is possible to constrain the "multiplicity" of such a mapping. For example, the relation CONTAINS implicitly provides a mapping from ship-cargo pairs to tonnages. For a given ship-cargo pair, there can be at most one tonnage (the amount of that cargo or that ship). But for a given tonnage, there is no restriction on the number of ship-cargo pairs which CONTAINS relates to it. This can be expressed by augmenting the declaration of CONTAINS:

relation CONTAINS (SHIP, CARGO, TONNAGE)
any (SHIP, CARGO) optional

The pair (SHIP,CARGO) defines the role partitioning. The other partition is implicitly (TONNAGE). The word any preceding the explicit partition indicates no constraint on the mapping from (TONNAGE) to (SHIP,CARGO). The word optional indicates that a given (SHIP,CARGO) pair may map to zero or one (TONNAGE). This constraint could also have been written

optional (TONNAGE) any

The ways to specify the multiplicity of a mapping include:

indicator	multiplicity
any	no restriction
optional	0 or 1
unique	exactly 1
multiple	1 or more
any integer N	exactly N

When attributes are used to specify binary relationships, it is still possible to state both forms of constraint within the type declaration which introduces the attribute. The "mapping multiplicity" constraints are indicated by placing multiplicity indicators before and after the attribute specification. For example:

type ship (unique REGISTRY | country any, ...) (2.3)

constrains the relationship between ships and their country of registry so that each ship must have exactly one country of registry, but a given country may have zero or more ships registered in it.

If no multiplicity indicator is given in a declaration the default is "unique" for the mapping from declaration type to attribute type, and "any" for attribute type to declaration type. Thus example (2.3) could be written simply as:

type ship (REGISTRY | country, ...)

defaulting both multiplicity constraints.

Constraints on individual relationships in an attribute relation can also be declared within the type declaration in which the attribute is declared. In fact, it is possible to constrain combinations of different attributes of a given type, as illustrated in the following examples:

```
type draft definition natnum
type port (MAXDRAFT | draft, COUNTRY multiple)
type ship (..., DRAFT.SHIP, any PORTOFCALL | port, PORT.HOME)
  where always prohibit draft.ship > portofcall : MAXDRAFT
  end
```

(2.4)

would prohibit a ship from having a port of call for which the ship was too large.

In addition, one could require that a ship be capable of entering its home port by including in the ship declaration

always require *draft.ship* $\leq$ *port.home* : MAXDRAFT

A type constraint of the form "always prohibit condition" declares that the condition may never hold for any instance of the type in whose declaration the constraint appears. The condition refers to the object being constrained by using the type name as a variable. It may also refer to the objects in attribute relationships with that object by the attribute names used in the declaration. The condition holds, and thus violates the constraint, if *any* combination of attribute values satisfy it. Thus constraint ((2.4)) prohibits a ship from having any port of call which is inappropriate regardless of how many acceptable ports of call it may have at the same time.

Analogously, an "always require condition" constraint on a type requires the condition to hold for every instance of the type, for all combinations of attribute values. In all cases, "always require condition" and "always prohibit ¬condition" are equivalent. As with other predicates, it is generally easier to read a constraint in a form which reduces the use of the logical operator ¬.

2.4.1 Use of Constraints

There are two primary purposes for constraints in Gist specifications. The first is as an aid to someone trying to understand or implement a (portion of) a specification. Understanding and implementation both involve a great deal of mental simulation of behavior. This in turn requires consideration of how behavior will proceed under various conditions. Constraints are a powerful means of limiting the range of conditions which must be considered.

The second use of constraints is to aid the specification writer. Consider two ways people use what we might term "constraints" in English--e.g, "Don't let your bank balance be negative."

- This might be treated as a rule to do something (although what to do is not stated) whenever the balance drops below 0 to rectify the situation. In programming terms, use of a constraint does not say that the condition always holds, but that when it doesn't hold, some form of "interrupt" process is used to re-establish it. Gist's semantics does not treat constraints in this way.

- Another use people make of constraints is to *limit choices*. The bank balance constraint will limit a person's selection when buying a stereo, for example. This is Gist's semantics for constraints--that choices are always to be made so that no constraint will ever be violated. This implies that choices are limited not only by their immediate affects, but by their "downstream" consequences as well. This is analogous to a person realizing that he can't buy a stereo, not because his bank balance would go negative when he brought it, but because it would go negative when he paid the rent two days later (a behavior for which he has no alternative).

It is often easier to specify behavior by allowing the procedural part of the specification to contain too many behaviors and letting constraints prune the alternatives.

In choosing what constraints are worth stating explicitly, a verbal process description is of limited use. Some constraints may be expressed verbally, but many are such "common knowledge" that they are left unstated--"two ships can't be in the same slip", "a ship may not be loaded above its capacity". A specifier can state any obvious constraints which come to mind before working on the procedural portion of his specification. But he should be on the lookout for situations where he is writing procedural specification for the purpose of "avoiding" a problem. An example of the way constraints can collaborate to simplify the procedural part of a specification is presented in section 2.8.

2.5 DERIVATION RULES

Derivation rules are provided in Gist to permit the specifier to expand his vocabulary to include concepts which can be defined in terms of other concepts, without having to treat the concepts as though they were independent and maintain their equivalence procedurally. We might say "a ship is docked at a pier if and only if it is berthed in some slip of that pier" to define the concept of "docked at" in terms of "berthed in" and the relationship between slips and piers.

Gist permits derivation rules for types and relations (including attribute relations). To define a new derived relation, the Gist specification should include a normal declaration of the relation, together with a predicate stating the conditions under which the relation holds:

```
relation TRADE-PARTNERS (COUNTRY.1, COUNTRY.2)
    definition TRADE (country.1, $.country.2) ∧
               TRADE (country.2, $.country.1)
```

would define a binary relation TRADE-PARTNERS which would hold between two countries if and only if countries had mutual TRADE relationships (regardless of the goods involved).

Derived concepts in Gist may be used in all ways like non-derived concepts (including their use in the deriveration of other concepts) save one: the derived relation may not be used as a primitive state change statement (see section 2.8.1).

A derived attribute is specified by including a normal declaration for it and a derivation for it as well.

```
type ship (..., optional BERTH | slip optional,
               optional DOCK | pier derivation ship : berth' : ATPIER
```

This defines the "DOCK" attribute to relate a ship to a pier if and only if the ship is in a slip at that pier.

Finally, one can give a definition for a type name, where the definition is a predicate, written in terms of the type name, which, if satisfied by some object, qualifies that object as an instance of the type.

```
type port (. . .) optional supertype of
    <oilport definition ∃ pier || pier : PORT = oilport ∧
                                  pier : HANDLES = Oil
    >
```

Under this declaration, a port is an oilport if and only if it has a pier which handles oil.[6]

2.6 OBJECT EXPRESSIONS

The *object expression* is Gist's means of referring to particular objects, or values, within predicates and statements. The referent(s) of an object expression is determined by the process state in which the expression is evaluated and the referents of any variables used freely within the expression. An object expression may refer to a single object, in which case it is said to make a *deterministic* reference. It may refer to multiple objects, in which case it is said to make an *ambiguous* reference. It is also possible for an object expression to have no referent at all, in which case it is said to make an *anomalous* reference.

2.6.1 Literals

The simplest object expression in Gist is a *literal*. Distinct literals refer to distinct objects, and all occurrences of a given literal refer to the same object. Both numeric literals (*5000*) and non-numeric literals (*Corn*) are allowed. Non-numeric literals must consist of single identifiers, and those identifiers should not be used as the names of relations, actions, types, or variables elsewhere in a specification.

2.6.2 Variables

Variable names in Gist are arbitrary identifiers. Each variable name has a type which restricts the potential referents of the variable. The type may be implicit in the variable's name, or may be declared explicitly. To have an implicitly typed variable, it is only necessary to choose its name in the same way as names are chosen for implicitly typed attributes--e.g., "*port.old*" could be used as the name for a variable with implicit type "port". Variables obtain their referents in two ways:

---

[6] A type which is declared to be a unique supertype of some collection of types may not also have a definition; the unique supertype of in fact constitutes a definition of the supertype.

- A variable can be used as a formal parameter of an action. It then obtains a referent object from an actual parameter in an invocation of the action, and can be used to refer to that object within the action's defining statement.

- Quantified variables can be used in predicates as they are in predicate calculus. These variables obtain as referents objects which satisfy the predicate. In many uses of predicates, these variable names can then be used later to refer to those objects.

2.6.3 Attribute Reference

Through the use of attribute names it is possible to refer to the objects related to a given object in an attribute relationship. Such an expression may be an anomalous, deterministic, or non-deterministic reference.

ship.1 : CAPACITY

would refer to the tonnage which is the capacity of the ship referred to by the variable "*ship.1*". ": attribute" maps objects of the type in which the attribute was declared (in this case, "**ship**") to objects of the type specified for the attribute (in this case, "**tonnage**").

USA :: REGISTRY

would refer to any ship whose country of registry was the USA. ":: attribute" performs the inverse mapping from ": attribute".

2.6.4 Descriptive Reference

A descriptive reference is a reference to an object obtained by specifying its type and, optionally, a predicate which must be true of it. Any object of the indicated type for which the predicate holds is then a referent of the expression. The expression

a *c* | **country** || TRADE(*c*,*Oil*,*USA*)

would have as its referent any country which traded oil to the USA. In this expression, *c* | **country** indicates that the type of object being referred to is "**country**", while "*c*" is an identifier used to refer to the desired object in the predicate which follows the symbol "||". This expression could also be written as:

a *country* || TRADE(*country*,*Oil*,*USA*)

An expression which refers to any instance of a type is written by omitting the predicate:

a *country*

2.6.5 Arithmetic Expressions

Expressions whose referents are arithmetic values may be combined using the arithmetic operators +, -, *, and \ to compute other arithmetic values, as is done in algebraic programming languages. Furthermore, the monadic operator "count" may be applied to any expression to obtain the number of referents which that expression has. For instance,

count(Seattle :: PORT.HOME)

is an arithmetic expression whose value is the number of ships having Seattle as their home port. If the operand of "count" is anomalous, then the value of the "count" expression is zero.[7]

Parentheses may be used to enclose subexpressions both for readability and to override the default groupings imposed by Gist's syntax.

2.7 PREDICATES

A *predicate* is the means by which conditionality is introduced into a specification. There are a number of basic pieces of information in a process state which can be used as the basis of tests:

- the existence of objects
- the classification of objects by type
- the relationships between objects

Like predicate calculus, Gist permits the expression of complex predicates through the use of quantified variables and logical combinations of predicates involving those variables and literals.

Semantically, a predicate may be said to be either TRUE or FALSE *relative to* a particular process state and particular assignment of referents to its variables. In the following descriptions, where no confusion will arise, we will simply refer to predicates as TRUE or FALSE.

2.7.1 Propositions

All predicates are built up from *propositions*. One form of proposition is a test for the *existence* of an object. Since objects may be created and later destroyed (see section 2.8.1) it is possible to have an expression (e.g., a variable) whose referent is an object which has been destroyed. The monadic predicate "extant" is used to test for the existence of the referent of an expression.

---

[7] Although using "count" to test whether the number of referents of an expression is zero does in effect test whether it is anomalous, the predicate "extant" (see section 2.7) is the stylistically preferred way of doing this.

extant *ship.1*

is a proposition which is TRUE if and only if the referent of the variable "*ship.1*" exists in the state in which the proposition is evaluated. Since a proposition using "extant" is FALSE if the operand of "extant" is anomalous, it is also useful for testing whether an expression has any referents.

Another basic proposition is a test for referential identity of two expressions. Any two object expressions may be compared with the logical operator " = ". The proposition is TRUE if and only if the two expressions have at least one referent in common. If either or both expressions is anomalous, or they have disjoint sets of referents, the proposition is FALSE.

ship.1 : PORTOFCALL = *ship.2* : PORT.HOME

would be TRUE if the referent of "*ship.1*" had the home port of the referent of "*ship.2*" as one of its ports of call.

It is also possible to test for non-identity of two expressions, using the logical operator "≠". The proposition is TRUE if the two object expressions have non-empty sets of referents which are not identical. If either or both expressions is anomalous, the proposition is FALSE.

ship.1 : PORTOFCALL ≠ *ship.2* : PORT.HOME

would be true if the referent of "*ship.1*" had any port of call other than the home port of the referent of "*ship.2*."

Another form of proposition is a test for the classification of an object.

Seattle :: PORT.HOME isa **oiltanker**

would be true if any ship having Seattle as home port was classified as an oiltanker.

Arithmetic expressions may be compared using the comparison operators <, ≤, >, ≥, and, of course, the identity comparison = .

The final form of proposition introduced in this section is a test for the existence of a named relationship between objects. The relation must be specified by name, and the objects being tested are specified by object expressions. A relationship is TRUE if any combination of referents of the object expressions fill the corresponding roles of any relationship in the named relation. If the named relation is n-ary, exactly n object expressions must appear in the relationship test. The correspondence between the object expressions in the proposition and the roles of the relation is positional.

CONTAINS(*Seattle* :: PORT.HOME, a *grain*, a *tonnage* || *tonnage* > *5000*) (2.5)

is a proposition which is true if there is any relationship in CONTAINS involving a ship whose home

port is Seattle, a cargo which is an instance of "grain", and a tonnage which is greater than 5000. Note the positional correspondence between the three object expressions in ((2.5)) and the roles in the declaration of **CONTAINS** in example ((2.1)).

An abbreviation which is frequently useful in propositions, and particularly in relationship tests, is the use of the symbol "$" as an object expression. It may be thought of as having all existing objects as its referents, and thus serves as a test of whether any existing object satisfies the predicate in which it is used. For instance,

CONTAINS(*Seattle* :: PORT.HOME, a *grain*, $)

is a proposition which is TRUE if any ship having Seattle as its home port contains *any* amount of a cargo which is an instance of "grain". It should be noted that, for each of the propositional forms, the proposition is TRUE if *any* referent(s) of the expression(s) used as operands have the property being tested, or bear the tested relation to one another. In all cases, the propositions have the semantics normally associated with them in programming languages when all expressions involved are deterministic.

2.7.2 Quantified Predicates

Predicates may make use of existentially and universally quantified typed variables. These variables appear freely within the scope of the quantifier. A predicate Q of the form:

$$\exists\, v_1|t_1,\ldots v_n|t_n \,\|\, P$$

is TRUE if and only if there exist objects $O_1, \ldots O_n$, where O_i is an instance of t_i, such that P is TRUE when free occurrences of v_i in P are taken to refer to O_i. Furthermore, Q is said to be TRUE *subject to* the assignment of O_i to v_i. Otherwise, Q is FALSE. For example.

$$\exists\, c \,|\, \textbf{cruiseship} \,\|\, c : \text{REGISTRY} = \textit{USA} \qquad (2.6)$$

would be TRUE if any instances of "**cruiseship**" had USA as their country of registry.

Analogously, a predicate Q of the form:

$$\forall\, v_1|t_1,\ldots v_n|t_n \,\|\, P$$

is FALSE if and only if there exist objects $O_1, \ldots O_n$, where O_i is an instance of t_i, such that P is FALSE when free occurrences of v_i in P are taken to refer to O_i. Furthermore, Q is said to be FALSE *subject to* the assignment of O_i to v_i. Otherwise, Q is TRUE. For example,

$$\forall\, o \,|\, \textbf{oiltanker} \,\|\, o : \text{PORTOFCALL} : \text{COUNTRY} = \textit{USA}$$

would be TRUE if every instance of "**oiltanker**" had some port of call in the USA. (It would also be TRUE, by definition, if there were no instances of the type "**oiltanker**".)

Quantified variables may, like role names, be named so that the variables type is implicit in its name. Example ((2.6)) could be written as:

$\exists$ *cruiseship* || *cruiseship* : REGISTRY = *USA*

2.7.3 Predicate Composition

Predicates may be combined with the unary prefix $\neg$ and the boolean infix operators $\wedge$, $\vee$, $\otimes$ (exclusive or), $\Rightarrow$ (implies), and $\equiv$ (equivalent) to form *logical compounds*. The precedence for parsing logical compounds is the standard precedence of propositional logic. Parentheses may be used for legibility and for groupings other than that given by the default precedence. The semantics of logical compounds is defined by the standard truth tables.

2.8 ACTION DECLARATIONS

Any application domain will have a variety of activities carried out by processes in that domain. These activities change the state of the process in some way. They may create or destroy domain objects (launching or decomissioning a ship), change object relationships (giving a ship an additional port of call), or classify or declassify objects (making a crewmember an officer). Furthermore, the execution of an action serves to "mark time" in the process, in the sense that Gist's facility for temporal reference permits reference to past execution states in terms of the start, completion, and duration of events (action executions). Thus an action which makes no changes to domain objects may still be a useful modelling concept.

The effect of an action on the process state depends on the state in which it is invoked and on the *parameters* used in the invocation. These parameters serve two purposes in Gist:

- As in conventional programming languages, the parameterization of a definition permits users of the definition to specialize it to suit their needs. In the case of an action, the user, or invoker, of the action can control its effects by specifying particular values (actual parameters) for the roles (formal parameters) of the action.

- Reference to events in Gist's temporal reference expressions is done through a combination of action names and parameter "descriptions". This permits a much finer filter on event reference than would reference by name alone.

An action declaration consists of a name for the action and a list of names (and types) for each formal parameter, or role, of the action. It also may contain a definition, and pre- and postconditions for the action. Figure 2-1 illustrates the declaration of actions.

```
action
 MOVESHIP[SHIP,PIER]
  precondition INPORT(ship,pier : PIERLOC),
  definition
    if ship : DOCK = pier
       then comment no movement needed end comment
       else update BERTH of ship to a slip
  postcondition ship : DOCK = pier;

action
 LOADSHIP[SHIP,CARGO,INCR : tonnage]
  precondition ship : DOCK : HANDLES = cargo,
  definition
   if CONTAINS(ship,cargo,$)
    then update TONNAGE of CONTAINS(ship,cargo,$)
                          to tonnage + incr
    else insert CONTAINS(ship,cargo,incr);

action
 ASSIGN-CARGO[CARGO,TONNAGE,PORT.FROM,PORT.TO]
  definition
  begin
   require ∃ship || ship : PORTOFCALL = port.to;
   MOVESHIP[ship,a pier || pier : PIERLOC = port.from];
   LOADSHIP[ship,cargo,tonnage]
  end
```

Figure 2-1: Action Declaration in Gist

The body of an action definition is constructed from the primitive *statements* of Gist and forms for combining them into complex process descriptions. The first action declaration defines an action named MOVESHIP. The action has two parameters, of types "ship" and "pier".[8] This action is intended to represent the activity of relocating a ship at a given pier in a port. The definition is a simple conditional. If the ship is already docked at the indicated pier, nothing happens; otherwise, the berth of the ship is "updated" to be some (unspecified) slip.

This is an inadequate definition of the intended activity, for two reasons. First, it fails to restrict which ships can be moved to a given pier. The intent is that MOVESHIP is describing activity *within* a port; it is not to be used to relocate a ship across thousands of miles. This deficiency is overcome by the use of a *precondition* that states the desired relationship between the ship and pier used as parameters--namely, that the ship must be in the port in which the pier is located.

The second problem is the non-determinism of the slip chosen as the new berth for the ship. Certain referents of the expression "a *slip*" will not achieve the intent of this action, which is to relocate the ship at the indicated pier. A more restrictive expression could be used (restricting the referents to slips at the desired pier) or, as in the example, the same goal can be achieved with a postcondition stating that, when the action is completed, the ship's dock must be the desired pier. In either case, it is unnecessary to incorporate in this action the restriction that only one ship may be in a slip at a time. That was stated as a global constraint on the attribute "BERTH", and thus serves to restrict the non-determinism in this action. Any attempt to invoke this action in a situation in which there are no empty slips at the desired pier would be *anomalous*, as would an attempt to use it with a ship and pier not satisfying the precondition.

The second action, LOADSHIP, is intended to capture the change of process state which occurs when cargo is added to a ship. In this case, all that is modeled is an update of the tonnage of the given cargo on the ship, so that the new tonnage in the relation CONTAINS is the sum of the previous tonnage and the increment loaded (or just the increment loaded if the ship did not contain any of the given cargo). To capture the intended restriction in our domain, a precondition requires that the ship be docked at a pier which handles the specified cargo.

Finally, the action ASSIGN-CARGO captures the process of getting a specified tonnage of a given cargo loaded onto a ship bound from one port to another. The body of this action begins with a

---

[8] The rules for naming and giving the type of formal parameters of actions are the same as the rules used everywhere else in Gist for variable and role names.

requirement that there be some ship having the target port as a port of call. It then uses the previously defined actions to move any such ship to any pier in the source port, and to load the cargo onto the ship. In this example, both the ship and pier are specified by descriptive references which are likely to have multiple referents. The global constraints together with the preconditions and postconditions of actions combine to restrict the non-determinism allowed by the definition alone. Thus:

- the ship selected would have to be located in the source port. (Precondition of MOVESHIP)
- the pier selected would have to handle the indicated cargo. (Precondition of LOADSHIP)
- the pier selected would have to have at least one unoccupied slip. (Global constraint on the "BERTH" attribute)
- the ship selected would have to be capable of carrying the indicated cargo (Global constraint(s), such as the non-mixing of fuel and grain cargos)

Clearly, the specifier has considerable leeway in Gist in defining the dynamics of a process. It can be done with the method used in programming languages--making every reference (sufficiently) deterministic within its local context so that the global behavior proceeds as desired. The specifier can also use greater non-determinism in his referring expression, but bound his action definitions with preconditions and postconditions which ensure proper global behavior. Finally, the specifier may use global constraints to restrict the non-determinism present in his action definitions. The remainder of this section describes the variety of constructs which may be used to compose action definitions.

2.8.1 Primitive Statements

The primitive statements of Gist permit classification and declassification of objects, addition and deletion of attribute values, addition and deletion of relationships, and creation and destruction of objects. In each case, the primitive statement specifies a transition from a state S to a state S', with *no* intervening states. In many cases, the primitive statement may involve some non-determinism, thereby specifying multiple possible state changes.

2.8.1.1 Object Creation

The create statement specifies the creation of new objects. A single create statement can specify creating one or more new objects, and establishing relationships among the new object(s) and existing ones. The new objects are created as part of a single state transition. In other words, the

create statement specifies a mapping from any state S to state S' that is identical to S except for the addition of the newly created objects, their classification into the indicated type(s), and the addition of the new relationships.

For example, the statement

create *ship* || *ship* : CAPACITY = *30000*,
ship : REGISTRY = *USA*,
ship : PORT.HOME = *Seattle*

would specify the creation of a new ship having a capacity of 30000 tons, a USA registry and a home port of Seattle. The new ship would not be involved in any other relationships initially.

2.8.1.2 Object Destruction

The destruction of an existing object is specified by another primitive statement in Gist. The "destroy" statement permits the specification, via an object expression, of some object to be destroyed. If the expression has multiple referents, then one of the referents is destroyed. If it has no referents, then the destroy statement is itself anomalous. A literal object may never be destroyed. Destruction of an object means eliminating all classification of it as an instance of any type, and removing all relationships involving it. This in turn means that a process has no way of referring to an object following its destruction except through variables that already referred to it prior to its destruction, or by temporal reference (see section 2.9). For instance:

destroy *Seattle* :: PORT.HOME

would (non-deterministically) eliminate some ship whose home port is Seattle. Following execution of the "destroy" statement, even the expression "a *ship*" would not have the destroyed ship as a referent.

2.8.1.3 Adding Relationships and Classifications

Addition of new information about existing objects is accomplished with an "insert" statement. The new information may consist of an additional classification for an object or a relationship between objects. As with the "create" statement, the relationships are specified by means of propositions. For example, the statement:

insert CONTAINS(*ship*,*cargo*.*incr*)

would add a relationship between the referents of the variables "*ship*", "*cargo*", and "*incr*" to the CONTAINS relation.

An additional classification for an object can also be added by the "insert" statement:

insert *port* isa **portofentry**

would classify the referent of "*port*" as an instance of the type "**portofentry**".[9]

2.8.1.4 Removing Relationships and Classifications

A relationship can be removed by means of the "delete" statement.

delete (a *ship*) : PORTOFCALL = *Seattle*

would remove (non-deterministically) some ship from having Seattle as a port of call.

"delete" may also be used, by analogy with "insert", to declassify an object as an instance of some type. An execution which would leave an object with no classification, however, would be anomalous.

2.8.1.5 Updating Relationships

Another useful change of state is the *update* of a relationship between objects. This has the effect of removing an existing relationship and adding a new one. The "update" statement specifies the relationship to be updated, by means of a proposition, the role in that relationship to be altered, and the new value to fill that role.

update BERTH of *ship* to a *slip*

would change the slip at which the ship referred to by "*ship*" was berthed to a non-deterministically specified slip.[10]

update TONNAGE of CONTAINS(*ship*,*cargo*,$) to *tonnage* + *incr*

would change the quantity of the cargo referred to by "*cargo*" on the ship referred to by "*ship*" to the sum of its previous value (referred to by the use of "*tonnage*" as a variable in the expression giving the new value) and the amount "*incr*".

If the relationship in an update does not hold, the update serves to add new information without deleting anything.[11] Conversely, if the relationship to be added already exists, the update statement serves solely to delete the old relationship. Finally, if the information to be deleted is identical to that being added, the update statement specifies no change to the process state.

---

[9] It is not necessary that additional classifications be subtypes of existing classifications, although that is the most common usage.

[10] The "new" slip could be the same slip at which the ship was berthed prior to the execution of the update statement.

[11] In this case, of course, the expression specifying the *new* object may not use the role name as a variable to refer to the previous object.

2.8.2 Action Invocation

A defined action is invoked by a statement which specifies the action name and an actual parameter for each of the action's formal parameters. Each actual parameter is specified by an object expression. The invocation is non-deterministic, allowing *any* referent of each expression to serve as the referent of the corresponding formal parameter. If any of the actual parameter expressions is anomalous, then the invocation statement is anomalous.

The invocation specifies the behavior specified by the action's definition.

LOADSHIP[*ship*,*cargo*,*tonnage*]

indicates the execution of the behavior defined by the action **LOADSHIP** using the referents of "*ship*", "*cargo*", and "*tonnage*" as actual parameters.

2.8.3 Compound Statements

Gist provides a variety of compound statements to permit specification of:

- Sequential state changes. A process encompasses multiple state transitions, not just a single transition from an initial to a final state.
- Conditional state changes. How a process alters a state may be conditioned by the state.
- Alternative state changes. The activity of a process may be non-deterministic in certain situations.

The statement syntax of Gist includes forms for specifying such state changes.

2.8.3.1 Conditional Behavior

The conditional statement permits selection of an activity based on the current process state. The simplest form of conditional statement consists of a predicate together with a statement to execute if that expression is TRUE and/or a statement to execute if it is FALSE. In addition to conditioning the behavior, the conditional statement may produce an augmented variable reference environment for the selected statement. If the predicate is TRUE subject to some assignment of values to variables (see section 2.7), the "then statement" is executed with that assignment augmenting the previously existing variable assignment. Analogously, if the predicate is FALSE subject to some assignment of values to its variables, the "else statement" is executed in an augmented variable context. Since there may be multiple distinct assignments subject to which a predicate is TRUE, or multiple assignments subject to which it is FALSE, the augmented assignment may be non-deterministic. If the predicate is TRUE, but the "then statement" is anomalous for all assignments which make it TRUE, the conditional statement is itself anomalous. Analogously, if the predicate is FALSE but the

"else statement" is anomalous for all assignments which make it FALSE, the conditional statement is anomalous. For example,

```
if ∃ ship || CONTAINS(ship, Wheat, a tonnage || tonnage ≥ 5000) ∧
        INPORT(ship, Seattle)
 then insert ship : PORTOFCALL = port
 else ASSIGN-CARGO[Wheat, 5000, Seattle, port]
```

would test whether there was a ship in Seattle which contained at least 5000 tons of Wheat. If so, some such ship would get the referent of "*port*" added to its ports of call. If not, the action ASSIGN-CARGO would be invoked to load wheat onto a ship in Seattle bound for that port. If there were one or more such ships in Seattle, but (because of constraints) it was not possible to assign the new port of call to any of them, then the if statement itself would be anomalous.

2.8.3.2 Sequential Behavior

As in most programming languages, the sequential execution of distinct statements is specified by a block statement, which is simply a sequence of statements bracketed by "begin" and "end". The lexical order of appearance of individual statements in the block specifies the order of statement execution, each statement being executed in the process state resulting from the execution of its predecessor.

2.8.3.3 Point Invariants

The primary restriction of non-determinism in Gist specifications arises from the various forms of constraint declaration, which preclude otherwise permissible executions. The declared constraints restrict non-determinism throughout their scope; they are not tied to any particular state of the process. *Point* invariants specify conditions which are required or prohibited only at particular states of the process. While point invariants appear in Gist as statements, they do not specify any state transition. They limit non-determinism in a specification by being *anomalous* if executed in a state in which the predicate is FALSE (in the case of a requirement) or TRUE (in the case of a prohibition).

Point invariants serve a second purpose in a Gist specification. When the invariant is satisfied, (one of) the variable assignment(s) subject to which the predicate was satisfied augments the existing variable assignment for the remainder of the lexical scope in which the point invariant appears. A given point invariant may be included in a specification for either of these two purposes--limiting non-determinism or augmenting variable assignment--or for both purposes.

For example, the point invariant:

```
require ∃ship || ship : PORTOFCALL = port.to
```

appearing as a statement would make anomalous any execution which reached that control point in a state in which no ship had the referent of "*port.to*" as a port of call. If there were such ships, however, there would be one continuation of the execution for each such ship. In each continuation, the variable "*ship*" would refer to the corresponding ship for the remainder of the lexical block in which the point invariant appears.

2.8.3.4 Alternative Behavior

Much of the non-determinism in a process is specified through use of non-deterministic object expressions in predicates and as actual parameters of actions. Non-determinism also results from the use of quantified variables in predicates which establish referents for those variables. However, non-determinism does not always factor so nicely, as in the English example "Either buy a second car or move closer to work". The *choice block* permits this kind of specification. The choice block specifies execution of any one of the statements in the block. The lexical order in which the statements appear has no significance. If every statement in the block is anomalous, the choice block is itself anomalous.

```
begin
 require ∃ ship || ship : PORT.HOME = Seattle:
 choose
   destroy ship;
   update PORT.HOME of ship to a port || port ≠ Seattle
 end choose
end
```

The above example specifies either of two possible transitions: a Seattle based ship is either to be destroyed or assigned a new home port.

2.8.3.5 Preferential Behavior

A situation which is similar to alternative behavior is one in which multiple alternatives are possible, but these alternatives can be ordered by preference. An English example is "Try to move closer to work, but if that isn't possible, buy a second car". This is expressed in Gist by a preference block. The preference block permits a specification to "control" the effect of anomaly. It specifies the execution of the first (in order of lexical appearance) statement in the block which is *not* anomalous. Which statement this is is in general dependent on the current process state. If every statement in the block is anomalous, then the preference block itself is anomalous.

```
begin
 require ∃ ship || ship : PORT.HOME = Seattle;
 attempt
   update PORT.HOME of ship to a port || port ≠ Seattle;
   destroy ship
```

```
    end attempt
  end
```

This example is identical to the "choose" block above, except that the order of statements in the "attempt" block is significant. In this case, we have specified a preference for reassigning the ship over destroying it.

2.8.3.6 Non-primitive Granularity

The domain model of typed objects and associations has a "basic" processing granularity. The primitive transitions at this level are insert, delete, update, create, and destroy. The ideal process being specified, however, may have a coarser granularity. That is, some conceptually indivisible state transition in the ideal can only be described in terms of multiple primitive transitions.

This issue must be faced in the specification language because domain constraints, temporal reference, and demon triggers are naturally defined with respect to states of the ideal process. But independent of this it is important to capture the granularity of the ideal process in the specified process. The primary reason for this is the enhancement of maintainability. Adding a new constraint or demon to a specification with the wrong granularity will not yield the desired new specification.

Rather than indicating when (particular) constraints and demons are to be checked, the specifier should define indivisible database transitions matching the granularity in the ideal process. The resulting specification will define a process having no spurious intermediate states.

In Gist, non-basic granularity is specified by enclosing multiple statements within "atomic" ... end atomic". The collection of individual transitions specified by the bracketed statements specifies a single, indivisible state change. For example, a ship's home port and country of registry could be simultaneously changed by:

```
atomic
  update REGISTRY of ship to country.new;
  update PORT.HOME of ship to port.new
end atomic
```

2.8.3.7 Iterative Behavior

The iteration statement consists simply of a "generator" of the objects over which the loop runs and a body, a statement which is executed on each iteration. There are two distinct types of iteration in Gist: iterations over the changing state of the data base and iterations over the state of the data base before the iteration begins.

To understand this distinction, consider the following (ambiguous) sentence: "Move ships so that every ship in port containing corn which is not at pier.2 is relocated there." The ambiguity (of interest in this section) is that of whether it is intended to achieve a state in which *all* ships containing corn are at the indicated pier, or if just those not originally there are to be relocated.

The basic distinction between the two iteration types is whether the predicate is applied to the initial state (before the iteration begins) or whether it applies to the (potentially altered) state at the start of each repetition. In the first case:

```
loop ∃ ship || INPORT(ship,port) ∧
               CONTAINS(ship, Corn, $) ∧
               ship : DOCK ≠ pier.2
 do
   attempt
    MOVESHIP[ship, pier.2];
    begin
      MOVESHIP[a ship, a pier];
      MOVESHIP[ship, pier.2]
    end
   end attempt
```

This iteration would (if possible) move all ships containing corn and not *initially* located at *pier.2* to that pier. They would be moved one at a time, in an unspecified order. If possible, a ship would simply be moved to the pier, without disturbing other ships. If necessary, however, some ship could be moved to make room for the new ship. Since the ship being moved could well be one which contains corn (including one moved to the pier on a previous iteration), there is no guarantee that the loop terminates with all ships containing corn at the pier. It is only guaranteed that each will have been at the pier during one of the iterations.

If the word "loop" is replaced by the word "while", the other iteration semantics obtains:

```
while ∃ ship || INPORT(ship,port) ∧
                CONTAINS(ship, Corn, $) ∧
                ship : DOCK ≠ pier.2
 do
   attempt
    MOVESHIP[ship, pier.2];
    begin
      MOVESHIP[a ship, a pier];
      MOVESHIP[ship, pier.2]
    end
   end attempt
```

In this case, a new ship is computed at the beginning of each repetition. The iteration will terminate as soon as, but not before, all corn carrying ships are docked at *pier.2*.

2.9 ADVANCED TOPICS

Gist contains several important specification concepts not covered in this introductory manual. In this section, the most prominent of these are mentioned.

2.9.1 Sets and Sequences

The concepts of *set* and *sequence* are primitive to Gist. Expressions can be written whose referents are sets or sequences of objects of some type. The elements of sets and sequences may themselves be sets or sequences. The need for sets is rarely felt in Gist, because non-deterministic expressions can be thought of as referring to "sets". Sequences, however, are needed to specify certain kinds of ordering information.

2.9.2 Multiple Lines of Control

Gist has a concept called "agent" which permits specification of processes which involve multiple lines of control. An agent is composed of a collection of actions, which specify that agent's "capabilities", and a collection of "demons", each of which specifies how the agent behaves in response to changes in the process state. For example, each port might be an agent in the ship domain, reacting to the arrival of ships at the port and to new purchase orders.

2.9.3 Temporal Reference

As a process executes, information is being produced and consumed. In writing a *program* to perform the process, a programmer must be concerned with the storage space required to hold this information. Programs manifest this concern by using compact or implicit representations of information, by representing only that information essential to correct execution, and, most pervasively, by releasing space used to store information that is no longer needed.[12] In a specification language, however, there is no reason to be concerned with storage space as a finite resource. As a process executes, the *current* collection of objects and associations changes, to be sure. But the history of execution and database states is conceptually well defined, in the sense that expressions and predicates can be assigned natural meanings with respect to *past* times as well as with respect to the current state. Gist permits reference to non-current information by means of

---

[12] Programming languages include facilities, such as block structure and garbage collection, which help the programmer deal with this storage allocation problem. More importantly, as we shall see, programming languages simply do not provide certain rich constructs, whose counterparts are available in natural language, that would make the storage allocation problem too difficult for current compiler capabilities.

reference to past process states. This in turn is accomplished by allowing predicates to be used to reference past states in which they were true, and by permitting reference to intervals of states which constitute *events*.

2.9.4 Boundaries

Gist permits the specification, by object type, of the "extent" of an object--that is, what other objects are an inherent part of the object. This permits the specifier to tailor, for each type, the semantics of high level concepts like destroy, object equivalence, and containment.

2.9.5 Orderings

Gist also permits the specification of "orderings" on objects. A rich sublanguage is provided for this, so the specifier does not have to "implement" the concept in terms of something more general. Once defined, the orderings can be used for comparing objects (just as numbers can be compared because they have a predefined ordering), and for controlling the execution order of iterations.

2.9.6 Meta Concepts

Collections of declarations may be named and parameterized, and then later specialized to a particular purpose. This permits the definition of high-level concepts like "labeled binary tree".

3. GENERAL APPROACH TO WRITING GIST SPECIFICATIONS

While it is not possible to give step-by-step instructions for the construction of a Gist specificaton, an outline of the usual approach should enable a Gist novice to avoid being overwhelmed by the task.

The basis of Gist is a simple underlying modeling structure: objects and relationships among them in the task domain are modeled by objects and relations in Gist (a relational database-like model). Change within the world is modeled by creation and destruction of objects, and by insertion and deletion of relations. Thus Gist specifications are fundamentally operational in nature. Gist's expressive power derives from the features built upon this underlying model.

To determine an appropriate set of object classes and relation names, a natural language description of the target process will prove useful. The description will be used only as a guide in the initial stages of development and need not be complete, detailed, nor even consistent.[13] This linguistic description will serve basically as a guide to the terms which need to be defined in Gist.

3.1 IDENTIFYING TYPES AND SUBTYPES

The first task of a specifier is to decide on a collection of object types that will capture the important objects, or values, that the process manipulates. A good rule of thumb is to consider common nouns (particularly concrete nouns) appearing in the linguistic description as the names for object types.

Although the collection of type names thus derived will likely need augmentation later, it should provide a solid basis for further development. At this time, it is also a good idea to identify any of these type names which are used to describe sets or sequences of other named types.

In any sizable application domain some names will refer to large classes of objects and other names to subsets of the same objects. Although often the verbal names used for types give no clue to these supertype connections, certain patterns should be considered. Noun pairs, such as "cruise ship", are frequently used to name subtypes of the second noun, as are compound nouns. Adjectives frequently are used with nouns in the same way, as in "military vessel". It is important to identify not only which classes include which others, but which sets of classes are disjoint and which overlap without strict inclusion. Often the use of Venn diagrams will prove helpful in this chore.

---

[13] For a serious software development project, of course, a good natural language specification is important. In this section, however, we are assuming that the specifier understands the process he wishes to specify quite clearly, but is unfamiliar with Gist.

3.2 IDENTIFYING INDIVIDUALS

In order to fully describe a process, it is sometimes necessary to refer to individual objects (values) as well as to entire classes. The names of individuals may be freely chosen by the specifier, but the type to which each belongs must be declared.

Linguistic clues that indicate the need for declaring individual instances of types include the use of proper nouns, mass nouns, and collections of adjectives (like color names) used to modify the same type names. These distinguished individuals are those which are used somewhere in the description of process activity or of constraints in the domain. They are not used for the purpose of describing "test data" for some test of specificaton behavior. A specification may have no such individuals. Even where they are necessary, it may not become apparent until later stages of development, when they are needed to describe the dynamic aspects of the process.

3.3 IDENTIFYING RELATIONS

The second major task of a specifier is to determine the interesting ways in which objects in the application domain can be related to one another. In describing a process in English, we use specific words and syntax to describe the ways objects are related to one another. In "ship bound for SantaBarbara" the phrase "bound for" indicates a particular relation that can hold between ships and ports--namely, a ship can be scheduled to go to a particular port. "A ship based in SantaBarbara" uses a different phrase, "based in", to talk about a different relation between the same two types. "A ship containing 50 tons of wheat" indicates yet another relation in this domain--ships may contain specific quantities of specific cargos.

In other cases the only linguistic indication of a relation will be a purely syntactic connection, such as the use of possessive noun phrases or prepositional phrase modifiers. The specifier must choose a distinct name for each distinct relation. It is generally wise to keep type names and relation names distinct. For each named relation, the specifier should identify the most general named types which can be related by that relation.

3.4 IDENTIFYING DERIVED CONCEPTS

The next step is to look for any identified relations which can be defined in terms of other named relations (possibly by recursive definition), and for identified types which can be defined in terms of other types and relations. These situations provide the basis for derivation rules in Gist. In some

cases, pairs of relations will be found where either could be defined in terms of the other. A particularly common case of this is where the two relations are inverses of one another. It is usually best to define both relations in terms of some more neutrally named relation.[14]

3.5 IDENTIFYING STATIC CONSTRAINTS

The declaration of types and relationships lays out a wide variety of possible process states--that is, states containing collections of instances of those types and relationships between objects of appropriate types. However, it is nearly always obvious that many of these states may never really arise. Sometimes this is because of physical constraints in the process domain. In other cases, it is not physical reality, but the desired process which restricts the potential states. In either case, it is wise to identify these constraints at this stage of development.

In choosing what constraints are worth stating explicitly, a verbal process description is of limited use. Some constraints may be expressed verbally, but many are such "common knowledge" that they are left unstated. A specifier can state any obvious constraints which come to mind before working on the procedural portion of his specification. But he should be on the lookout for situations where he is writing procedural specification for the purpose of "avoiding" a problem. A particularly common kind of constraint which can generally be identified at this stage is a "cardinality" constraint. For each binary relation R relating types T1 and T2 (not necessarily distinct), consider how many instances of T2 may be related in R to a single instance of T1, and vice versa. Sometimes there will be only a lower bound on the number, sometimes an upper bound, and often both. When a relation is a function, the upper bound will be 1. If it is total, the lower bound will be 1. One should also determine if there is a more restrictive cardinality restriction on R when considering subtypes of T1 and T2.

3.6 IDENTIFYING ACTIONS

The information gathered in these steps provides the basis for expressing, in Gist declarations, the static structure of the application domain. Any application domain will also have a variety of activities carried out by processes in that domain. These activities change the state of the process in some way. They may create or destroy domain objects, change object relationships, or classify or declassify objects. Furthermore, the execution of an action serves to "mark time" in the process, each action execution being an "event" in the process history. The linguistic process description can

---

[14] Ultimately the entire specification may be conveniently written in terms of the more neutral relation without any reference to the derived terms

serve as a guide to the actions which need to be defined. In general, transitive verbs and nominalized verbs indicate some action which needs to be defined. For each action, the specifier should determine the types of objects on which the action can operate, any "preconditions" on such objects which restrict the situations in which it is appropriate to perform the action, and a statement of how the action changes the process state.

3.7 IDENTIFYING DYNAMIC CONSTRAINTS

A more difficult class of constraints on a process consists of conditions restricting allowable *change* in the process state. Identifying these, like identifying static constraints, can make the specification both easier to understand and easier to construct correctly. The simpler forms of such constraints to look for are restrictions prohibiting change in certain relationships involving objects following their creation, or restricting the nature of such change.

3.8 IDENTIFYING ACTIVE PARTICIPANTS

In most large systems, it is natural to think of the behavior as being produced by the interaction of multiple participants, or *agents*. These agents are often themselves objects in the application domain, and thus can be classified by the same means used for non-active objects. For each agent class, the specifier should attempt to identify the named actions which are performed by agents of that class.

3.9 PROGRESSING IN COMPETENCE

After following this approach in the specification of a few simple processes, the specifier will become sufficiently familiar with the fundamental concepts of Gist to intermix the gathering of information about his application domain with the encoding of that information in Gist. He should also feel free to be less rigid about the order in which information is gathered and encoded. These steps are illustrated in the next chapter, particulary in the first example.

4. CONSTRUCTING GIST SPECIFICATIONS

In this chapter we wish to illustrate our approach to writing Gist specifications (as described in the previous chapter) through three examples. These examples are all real, moderate-sized systems chosen to demonstrate the applicability of Gist to a wide range of military applications. For each of these examples we describe the system to be specified and how we applied the Gist approach to specification writing to formulate these specifications. The description of the application of our approach to specification writing is most detailed for the first example.

The actual Gist specifications for these examples can be found in Appendix II.

4.1 SOURCE-DATA MAINTENANCE EXAMPLE

The task we are about to specify was chosen to be a test of the existing features of Gist, to suggest the need for additional features, and to promote development and recognition of principles of "good practice" in the construction of specifications. The size of the task, whilst still small in comparison to many real-world activities, is larger than "toy" examples. As such, we begin to see how the difficulty of constructing and understanding specifications in Gist grows with the scale and complexity of the task.

The example we specify is taken from [4]. It is concerned with maintaining programmers' source code, where the code is arranged into a hierarchical structure of units, files, libraries and projects. The user issues a batch sequence of commands to create, modify, and destroy portions of this structure. Figures 4-1 and 4-2 summarize the structure of the domain and the user commands to be supported.

The description upon which we base our specification consists of 16 pages of English text, plus 9 pages of "HIPO" diagrams which graphically depict input, functional processing, and output of the system. Our intention was not merely to formalize the English description, but rather to specify formally the desired behavior of the source-data maintenance system implied by the English. The crucial difference is that we regard the English description as being biased in many places towards a particular implementation. The disadvantages of this bias are that it compromises one's ability to perceive the desired behavior as opposed to the chosen implementation, and as a result some portions of the English specification are wrong (in the sense that although it is a specification, it is clear that the behavior implied is not the behavior intended), some are ambiguous, and some important details are left unstated, and that it overly constrains the implementation.

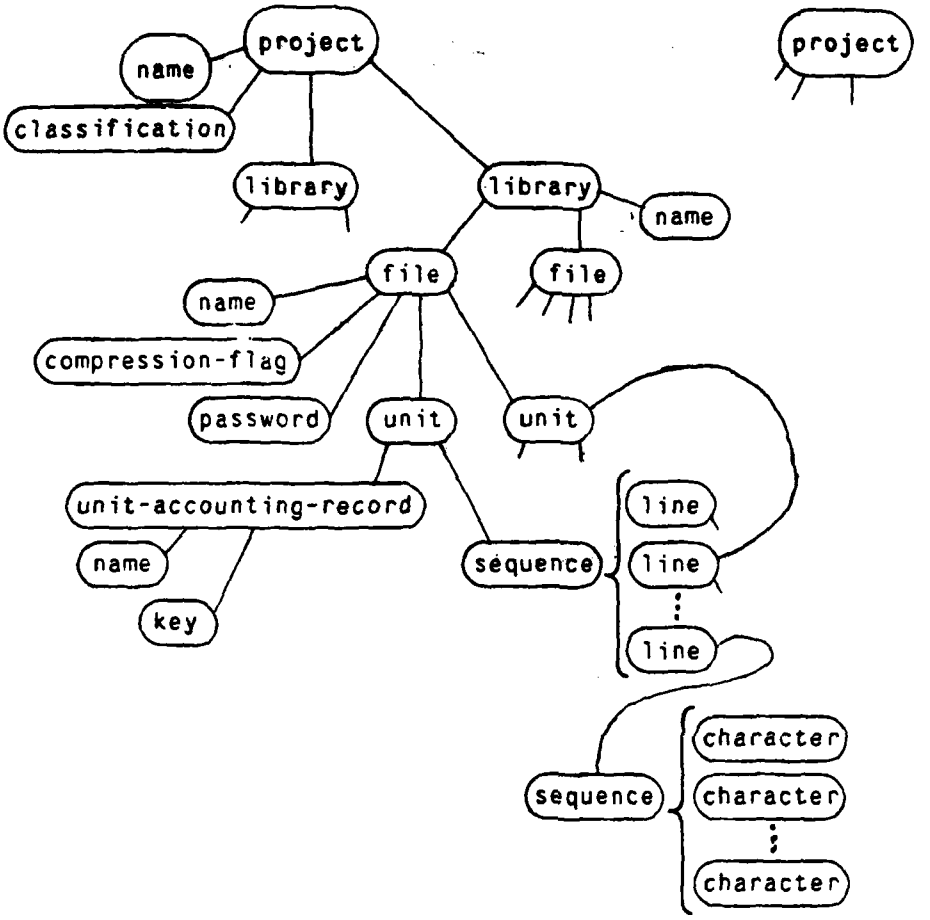


Figure 4-1: Structure of source-data maintenance domain

4.1.1 Suppressed details

Before we commence description of our specification, we make explicit those features we have chosen not to specify. These fall into three classes:

1. Implementation details. From our earlier comments, it comes as no surprise that we should suppress details of this nature. Examples of such suppression include

   - Data compression by removal of blanks when writing code to disc (this would be a constraint upon an implementation for an environment short of space);

   - Unit Accounting Records which package together information about units (in fact they are merely a convenient implementation package, not objects that the user observes).

   Of course, when directing the implementation of a specification we will wish to constrain the implementation choices in some ways. To do this we would provide such constraints

```
  command name                purpose

UPDATE                        Maintenance of existing files

subcommands of UPDATE:
   ADD                        Add a new unit
   PURGE                      Remove a unit
   REPLACE                    Replace the contents of a unit
   COPY-UNIT                  Copy a unit
   CHANGE                     Maintenance of existing units

   subcommands of CHANGE:
      INSERT                  Insert lines into unit
      DELETE                  Delete lines from unit
      SHIFT                   Shift characters left or right on line(s)
      MODIFY                  Modify line(s)
      COPY-LINES              Copy lines into unit
      + explicit insertion/deletion of numbered lines

COPY-FILE                     Copy file across project

BACKUP                        Preserve for later restoration projects,
                              libraries, files or units

RESTORE                       Restore preserved state of projects,
                              libraries, files or units
```

Figure 4-2: User commands supported by the system

separate from the specification of the functional behavior, which itself leaves maximum freedom of choice to the implementor.

2. Interface details. We aim to specify the desired behavior independent of the interface with the external world. This is accomplished by modeling the relevant portions of the external world as another (incompletely described) participant with whom interactions occur via the standard internal communication mechanisms. As part of the implementation of such specifications, external interfaces are chosen and the mappings between the external and internal representations defined. Such concerns should not compromise the clarity of the functional specification. This leads to the suppression of

   - the precise format of incoming commands (we assume that we receive a sequence of objects of type command);
   - the identification of objects in the world by alphanumeric names (we assume that the commands provide a reference to the object they are associated with, just as all modeled objects provide references to other objects through relationships);
   - identification of lines by sequence numbers (again, we assume that the commands reference the lines in question directly).

Naturally, any implementation will obviously have to take into account these requirements, which we would state separately.

3. Refinements. We concentrate upon specifying the "normal-case" behavior of our world. For example, although our specification must trap all error conditions, explicit provision of diagnostic error messages for all the cases is detail we wish to suppress for the time being, and would provide as a separate refinement of the normal-case behavioral specification.

4.1.2 Plan for Constructing Source-Data Maintenance Specification

As described in the previous chapter on writing Gist Specifications, a specification is organized to present the features of the task being modeled iin a comprehensible manner. This organization is repeated below, particularized by the features of the source-data maintenance domain.

1. Define the object types and relations to model the objects of the domain (projects, libraries, files etc.) and the relationships amongst them (e.g., files are related to libraries). As we make these definitions we may conveniently express some of the constraints of the domain (e.g., that every file belongs to precisely one library).

2. Define the "static" constraints on our domain--static in the sense that these constrain the state of objects and relations at every point in time without reference to their history (e.g., lines in units are of length Card-Length).

3. Define the types and relations to model the internal interface with the environment (user commands).

4. Define the actions which change the state of the world.

5. Define the "dynamic" constraints on our domain--dynamic in the sense that these constrain how the world may change from state to state.

6. Model the active participants of the domain who react to certain stimuli in the world and respond by invoking actions to appropriately modify the world.

This organization is convenient both as a framework to aid understanding of the specification and as a framework to introduce the Gist language features.

4.1.2.1 Modeling objects and relationships of domain

Figure 4-3 illustrates the modeling we aim to achieve at this stage. Objects and relationships among them within the task domain are modeled as objects and relationships in the specification domain. Files, libraries, etc., are modeled as distinct objects connected by relations. This simple yet powerful basis allows us to model task domains in a very direct fashion.

The objects are typed, so our first definition is that of the types of objects of the domain, after which we will define the relations linking them. E.g.:

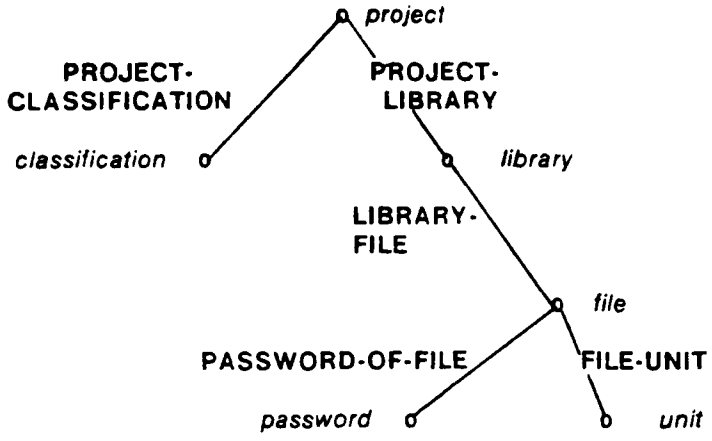


Figure 4-3: Modeling of objects and relations in Gist

```
type project;  type library;  type file;  type unit;

relation  PROJECT-LIBRARY(PROJECT | project , LIBRARY | library);
```

defines PROJECT-LIBRARY to be a relation between an object of type **project** and an object of type **library**. Each position in the relation definition is filled by <identifier> | <type>. where identifier is an arbitrary name. By convention, when *the* identifier is the name of the type, the identifier alone suffices. E.g.:

```
relation PROJECT-LIBRARY(PROJECT,LIBRARY);
```

The relations are not restricted to be binary, nor do they have any inherent directionality (i.e., it is possible via PROJECT-LIBRARY both to derive from a project a library, and to derive from a library a project).

4.1.2.1.1 Notation for types and binary relations

Often we have occasion to define many binary relations, as is the case in our example. For this purpose a convenient notation permits the simultaneous definition of a type and binary relations it participates in. E.g.:

```
type  project(LIBRARY,CLASSIFICATION);
```

declares **project** to be a type. and declares two binary relations which connect projects with libraries and projects with classifications. We refer to LIBRARY and CLASSIFICATION as the attributes of **project**. Note that there is a degree of freedom in choice of which type to make the attribute of the others; in the above case we could just as easily have made project be an attribute of libraries. This is purely a matter of taste (remember that we are using a notational convenience for the underlying type and relation definitions, which themselves have no preferred directionality) and should be made to

most closely conform to the accepted conceptualization of the domain (i.e., improve understandability). The notation supports extraction of information through attribute relations, e.g.:

to denote a classification related to project *p* (i.e., going downhill in Figure 4-3) we would write
p:CLASSIFICATION

similarly, to denote a library related to project *p* (again going downhill in Figure 4-3) we would write
p:LIBRARY

but, to denote a project related to library *l* (i.e., going uphill in Figure 4-3)
l::LIBRARY
("::" to go from attributee to attributor)

The "::" notation is easiest to remember in terms of the following rule
if *p*:LIBRARY = *l*
then
l::LIBRARY = *p*

The "::" notation should be read as "the object which has *l* as its LIBRARY attribute".

Last, we may also incorporate into this notation some simple constraints on the participation of objects in such relations (over and above the implicit constraint by type). For example, within our domain every project is related to exactly one classification. Similarly, every library is related to exactly one project. However, a project may have any number (including 0) of libraries related to it. These constraints--all restricting the number of objects that may participate in relations--can be expressed within our type/attribute notation. To constrain how many objects of the attribute type may be attributed to a single object of the defined type, we follow the attribute name with ":" and an appropriate keyword. Similarly, to constrain how many objects of the defined type may share as an attribute a single object of the attribute type, we follow the attribute name with "::" and an appropriate keyword. Our choice of keywords includes any (no restriction), unique (precisely one), multiple (one or more), and optional (zero or one). E.g.:

```
type project( LIBRARY :any ::unique , CLASSIFICATION :unique ::any );
```

(The use of ":" and "::" is intended to parallel their use in expressions, so just as "*p*:LIBRARY" denotes a library attributed to *p*, "project(LIBRARY :any ..." constrains how many libraries may be attributed to a project.)

One final remark before we show the definitions for our example is to state defaults for these constraining keywords: we have chosen to let omission of ":" and following keyword to be equivalent to :unique, and omission of "::" and following keyword to be equivalent to ::any.

4.1.2.1.2 Type and attribute definitions for the domain

We define here the types and attributes for the domain of the example specification:

```
type project( LIBRARY :any ::unique , CLASSIFICATION);

type library( FILE :any ::unique);

type file( UNIT :any ::unique , PASSWORD :optional);

type unit(UNIT-CONTENTS | sequence of line , UNIT-KEY | key :optional ,
                SOURCE-LANGUAGE | language :optional);

type line(CHARS | sequence of character , LINE-INCLUDES-UNIT | essence of unit :any)

type language(); type key(); type classification(); type password();
```

Two remarks on the above:

Notation: sequence of is a built-in type constructor. An object of type "sequence of t" (where t is some type) is a sequence of objects of type t. In the following examples we shall see notation to allow construction, modification and interrogation of sequences. Other type constructors are set and multiset. Sequences, sets, and multisets are included in Gist because of the convenience they provide. They are built-in parameterized types. In contrast, user-defined types may not be parameterized in Gist. This restriction does not appear to be a drawback, at least in the examples we have considered. We believe our specification style circumvents the need for such power, since our objects are atomic, gaining their definition from their relation to other objects. This is against the trend of modern typed languages, in which the user objects are complex entities having structure and values.

The LINE-INCLUDES-UNIT attribute of line models the ability to include notionally a unit at that line's location by referencing the unit (so that for compilation, listing etc., the effect is as if the contents of the referenced unit were included at that point). Because the source-data maintenance system is intended to support top-down programming, it must be possible to "include" a unit which does not yet exist. Gist permits references to objects that may not yet exist (or may no longer exist) by declaring such references to be of type "essence of t", where t is the type of the object being referenced. as used above in the type definition for line.

4.1.2.2 Static constraints

We use constraints in specification to state concisely and explicitly all the constraints that the model, in reflecting the world, must satisfy. We saw in the previous section some simple constraints on how objects could fill relations. A more general language construct permits the use of any predicate as a constraint which must always be satisfied. If we think about implementation, this may imply considerable expense in checking constraints every time the world changes in a way that might violate them. In implementing constraints we probably would want to distribute appropriate checks throughout the code at all potential violation points. During specification, however, it is better to state the constraint just once; and this aids understandability, reliability and maintainability (having stated a constraint, we are assured that the specified behavior must always satisfy that constraint. even if we modify or introduce new activities).

4.1.2.2.1 A "static" constraint

A constraint of the source-data maintenance world is that

"All lines within the unit-contents of a unit must be of length Card-Length."

This we express by

```
always required
    ∀ line || ∃ unit, integer || MEMBER(unit:UNIT-CONTENTS, integer, line)
                                  ⇒ ( length(line:CHARS) = Card-Length );
```

This exhibits some of our notation in use. We follow the convention that

- underlining denotes "reserved words" of Gist
- **bold lower case** denotes type names
- SMALL CAPITALS denotes attribute names
- *lower case italics* denotes variable and parameter names
- BOLD UPPER CASE denotes names of relations, actions and derivations
- *Mixed Case Italics* denotes objects referred to literally

In the above example, always required introduces a predicate which must be true in every state of the world. The predicate in this case is a quantification over all objects in our world of type line. MEMBER is a ternary relation among a sequence, a location within that sequence (specified by a positive integer) and the object at that location. Thus for all lines in our world, for those that are in the unit-contents (a sequence of lines) of any unit, their character sequence (extracted by taking the :CHARS attribute) must be of length *Card-Length*.

We have a shorthand which simplifies writing a predicate of the form ∃ <variable> || ...<variable>... where the variable occurs only once in the scope of the existential. The shorthand consists of omitting the "∃ <variable> ||", and writing $ in place of the variable in the body. So the body of the above constraint becomes

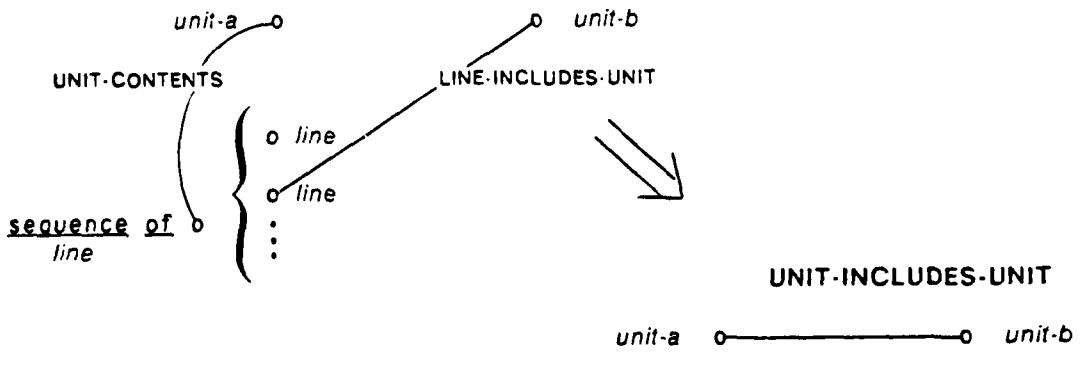
```
∀ line || MEMBER($:UNIT-CONTENTS, $, line)
        ⇒ ( length(line:CHARS) = Card-Length )
```

4.1.2.2.2 The use of derivation

A second example of a constraint in our world concerns the "line-includes-unit" relation, the means by which a line within the unit-contents of a unit may reference another unit. The intended use of this is that during compilation, listing etc., the contents of the referenced unit be included at the location of the line. The constraint we wish to impose is that such inclusion may not be circular (with the obvious meaning).

To express this constraint, we build in stages. First, for convenience, we define a new relation UNIT-INCLUDES-UNIT which relates unit-a to unit-b whenever unit-a contains a line "including" unit-b. Second, we form the transitive closure of UNIT-INCLUDES-UNIT. Last, we express the constraint by simply prohibiting self inclusion (since any circular chain will have led to derivation of UNIT-INCLUDES-UNIT between a unit and itself). To define the new relation we make use of a Gist construct called a derivation, which is a means of automatically inferring some relation to hold whenever a predicate holds. E.g.:

```
relation   UNIT-INCLUDES-UNIT(INCLUDOR | unit , INCLUDED | unit) ;

derivation  DERIVE-UNIT-INCLUDES-UNIT(unit-a | unit , unit-b | unit)
   whenever  ∃ line || MEMBER(unit-a:UNIT-CONTENTS, $, line) ∧
                                   ( line:LINE-INCLUDES-UNIT = unit-b )
   conclude  UNIT-INCLUDES-UNIT(unit-a,unit-b) ;
```



To form the transitive closure of UNIT-INCLUDES-UNIT we say

```
derivation DERIVE-TRANS-CLOSURE-UNIT-INCLUDES-UNIT(unit-a | unit , unit-c | unit)
   whenever ∃ unit-b | unit  || UNIT-INCLUDES-UNIT(unit-a,unit-b) ∧
                                UNIT-INCLUDES-UNIT(unit-b,unit-c)
   conclude  UNIT-INCLUDES-UNIT(unit-a,unit-c) ;
```

Finally the constraint may be expressed by

```
   always prohibited
      ∃ unit || UNIT-INCLUDES-UNIT(unit,unit);
```

The use of derivations is akin to the use of constraints insofar as it allows the explicit statement of the derivation rather than distributing the derivation mechanism throughout the code where it might be appropriate.

4.1.2.3 Modeling user commands

We model user commands causing appropriate changes to the source-data by making the commands into explicit objects. The user provides a sequence of such commands which the system then processes, performing the appropriate action(s) for each command. An alternative would have been to define actions for each of the commands and expect the user to invoke the actions directly (we would then have had to impose constraints on the order in which commands could be issued--e.g., an add command must have been preceded by a corresponding update command). Our choice was motivated by the nature of the described task--the system acts in a batch-like manner, accepting a sequence of commands which it then processes.

An implication of this choice is that many of the attributes of commands must be of type "essence of t" rather than simply "t", because the command exists before or after the object referred to (e.g., one of the update-sub-commands in a sequence of such may be a purge-command, which will continue to exist beyond the point of its processing; however, the unit will have been destroyed).
The command type definitions are as follows:

```
type file-spec(FILE, PASSWORD :optional);
                (we introduce file-spec to connect a file reference with a password)

type command()  supertype of
              < update-command ; copy-file-command ; backup-command ;
                restore-command >;

type     update-command(CURRENT-FILE-SPEC | file-spec ,
                        UPDATE-SUB-COMMANDS | sequence of update-sub-command);

type        update-sub-command(CURRENT-UNIT | essence of unit,
                               UNIT-KEY | key :optional)
                  supertype of
```

```
                    < add-command ; purge-command ; replace-command ;
                                  copy-unit-command ; change-command >;

type        add-command(DATA | sequence of line,
                        SOURCE-LANGUAGE | language :optional) ;

type        purge-command();

type        replace-command(DATA | sequence of line);

type        copy-unit-command(FROMUNIT | essence of unit,
                              FROM-FILE-SPEC | file-spec :optional);

type        change-command(CHANGE-SUB-COMMANDS |
                           sequence of change-sub-command);

type        temporary-change-command() subtype of < change-command >;

type           change-sub-command supertype of
                     < insert-command ; delete-command ;
                       shift-command ; modify-command ;
                       copy-lines-command ; replace-line-command >;

type              insert-command(DATA | sequence of line,
                                 SEQUENCE-LOCATION :optional);

type              delete-command(LINES | essence of sequence of line);

type              shift-command(LINES | essence of sequence of line,
                                COLUMNS | integer);

type              modify-command(LINES | essence of sequence of line,
                                 NEW-STRING | sequence of character,
                                 STARTING-COLUMN | integer :optional,
                                 OLD STRING | sequence of character :optional);

type              copy-lines-command(FROMUNIT | essence of unit,
                                     LINES | essence of sequence of line,
                                     SEQUENCE-LOCATION,
                                     FROM-FILE-SPEC | file-spec :optional);

type              replace-line-command(OLDLINE | essence of line, NEWLINE | line);

type    copy-file-command(TO-FILE-SPEC | file-spec, FROM-FILE-SPEC | file-spec);

type    backup-command(OBJECT | essence of project ∪ library ∪ file);

type    restore-command(BACKUP-STATE | state , OBJECT | essence of
                                project ∪ library ∪ file ∪ unit);
```

4.1.2.4 Modeling change

The simple and unified modeling of the world as a set of objects and relations among them permits a simple model of change. The primitive actions that cause change are:

- create - create a new object of a given type
- destroy - destroy an existing object (and in so doing delete all relations in which it participated)
- insert - insert (in the state) a relation among objects
- delete - delete an existing relation among objects
- update - change the object filling a role in some relation

These serve to build up all the changes we wish to model.

4.1.2.4.1 Object boundaries

Some conceptually simple and common activities in the world may involve many primitive actions to accomplish them, e.g., when deleting a unit we may wish the sequence of lines attributed to that unit to be deleted also. Languages which use structured object definitions also use that structure to determine the extent of such operations. We feel this is a serious overloading of concepts. Instead, we explicitly define the extent or "boundary" of our structural objects so that actions may be applied to the entire collection of objects and relations associating them.

To do this we augment type definitions with boundary definitions, which serve to direct how operations such as destroy, create, etc., are to affect an object and its relationships. E.g.:

```
type unit( ... ) bound equiv by
                        equiv :UNIT-CONTENTS,
                        =     :UNIT-KEY,
                        =     :SOURCE-LANGUAGE ;
```

tells us that two units are equivalent if their UNIT-CONTENTS attributes are equivalent (a recursive application of equivalence), their UNIT-KEY attributes are identical (i.e., if either has a key as such an attribute, then they both must have one, and it must be the same key--equivalence of keys is not sufficient) and their SOURCE-LANGUAGE attributes are identical. The use of "equiv" or " = " before the attribute name distinguishes whether to apply equivalence testing recursively between that attribute of each object, or simply test the attribute values for identity.

The same boundary definition will not necessarily be appropriate for different operations; e.g., in

destroying a unit, we may wish destroy to be applied recursively to the UNIT-CONTENTS attribute but not the UNIT-KEY or SOURCE-LANGUAGE attribute, in which case our boundary definition for destroy would read:

```
type unit( ... ) bound destroy by :UNIT-CONTENTS ;
```

note that for the purposes of destroy either we follow an attribute and recursively apply destroy to the related object or we do not; there is no distinction corresponding to the equivalent/identical distinction as appeared in defining equivalence between two objects. Hence in defining a boundary for destroy we simply name the attributes to be followed without prefixing them with either "equiv" or "=".

4.1.2.4.2 The top-level action to support user-commands

We define in our model an action SOURCE-DATA-MAINTENANCE which takes as argument a sequence of commands and makes the appropriate changes to the source-data as directed by those commands. We specify the interface to the user by defining as part of the system a demon (a construct with two parts, a trigger, a predicate, which upon becoming true causes the response to be performed). Its trigger would be the user activity to initiate processing; its response would be the invocation of SOURCE-DATA-MAINTENANCE with the appropriate command sequence. This interaction paradigm serves to clearly delineate the interface boundary between user and system, and is general enough to describe interfaces between multiple, independently active participants. In our simple case an implementation would undoubtedly utilize a simple subroutine call as interface.

The definition of SOURCE-DATA-MAINTENANCE is as follows:

```
action
   SOURCE-DATA-MAINTENANCE[commands | sequence of command]
   definition
      over commands named com
      do attempt
            case com of

               update-command =>  UPDATE[com];

               copy-file-command =>  COPY-FILE[com];

               backup-command =>  BACKUP[com];

               restore-command =>  RESTORE[com];

            end case ;
```

The construct over <sequence> named <variable> do <statement> iterates through the sequence in

order, binding the variable to the current element of the sequence within the scope of the statement. Here the effect is to invoke the appropriate lower level action depending upon the sub-type of the command.

The most interesting part of this definition lies in the use of the attempt construct. Within Gist, if a transition leads to a state that violates a constraint, we say that the resulting state is anomalous, and prohibit that transition. In some cases there may be more than one way of performing the transition (e.g., if we said

begin require ∃ *unit* ; destroy *unit* end

and several units exist, then we may pick any one of them to be destroyed). If all possible ways of performing a transition from some state lead to anomaly, then that state itself is anomalous. Thus anomaly can propagate backwards. Attempt is a construct that limits such propagation. We use it here in a simplified form of its more general version,

attempt <statement$_1$> then <statement$_2$> else <statement$_3$>

This has the semantics of <statement$_1$> ; <statement$_2$> if there is a possible non-anomalous execution of <statement$_1$>, otherwise <statement$_3$>. We use attempt in our example to limit the failure of a user-command to that command alone, rather than aborting the processing of the entire sequence of commands.

Definition of action UPDATE *is similar to* SOURCE-DATA-MAINTENANCE. It is an iteration through a command sequence, invoking the appropriate lower level command. We now turn our attention to the actions to perform the lowest level commands.

4.1.2.4.3 A simple action

The purge command (to get rid of a unit) is supported by the following action:

```
action  PURGE[unit]
   definition  destroy unit ;
```

The simplicity of this definition is due to two factors. First, the separate statement of constraints means we need not clutter PURGE's definition with checks to ensure that the user has provided the correct unit-key (as required in this example for those units which have an optional unit-key), that the unit is not included elsewhere, etc. Second, the separate definition of a destroy boundary for type unit ensures that when we state destroy *unit*, we also destroy its UNIT-CONTENTS sequence of lines.

4.1.2.4.4 Adjusting process "granularity"

The ADD command (to add a new unit) is a little more complex:

```
action  ADD[unit | essence of unit, file, add-command]
   definition
   begin
      atomic
         create unit, copy-of-lines | sequence of line ||
                                      copy-of-lines ≡ add-command:DATA   ∧
                                      unit:UNIT-CONTENTS = copy-of-lines;

         insert file:UNIT = unit
      end atomic;
      if ∃ key || key = add-command:UNIT-KEY
      then insert unit:UNIT-KEY = key
   end ;
```

The construct create *unit* ... || ... *unit*:UNIT-CONTENTS = ...; is used to create a new unit and provide it with its UNIT-CONTENTS attribute.

The interest here lies in the use of the atomic <statement>; ...; <statement> end atomic construct. The problem is that we must create a unit and insert it as an attribute of a file. Because of our constraints, all units must be attributed to a file; hence we must somehow perform the creation and insertion within a single state transition, so that on completion all will be well. This is precisely what atomic provides--the enclosed statements are all executed conceptually within a single state transition. so that constraints are checked only at completion, not during spurious intermediate states. (If the state resulting from an atomic is anomalous, then the whole transition is anomalous.) Thus atomic is a means of increasing the processing "granularity" to a coarser size than the primitive transitions (create. destroy. insert. delete and update).

4.1.2.4.5 Non-determinism and constraints

Non-determinism is a common feature of Gist specifications. Specification of a particular object may be non-deterministic, e.g.,

begin require ∃ *variable* | type ; <statement>; <statement>; ... end

binds the variable to any object of the named type within the scope of the following statements. (If no such object exists, it is an anomaly.) Specification of the order in which to do some actions may be non-deterministic, e.g.,

over { <object> , ... , <object> } do <statement>;

iterates through the set of objects in any order.

The task we are specifying may be non-deterministic in nature (but this is not the case with the source-data maintenance task). This is not the sole use of non-determinism; often it is particularly convenient to specify some activity in a non-deterministic manner and let the constraints filter-out only the acceptable paths. This allows us to specify the desired behavior without having to algorithmically determine what choice necessarily must be made. This latter form of non-determinism does occur within our specification, as we shall see next.

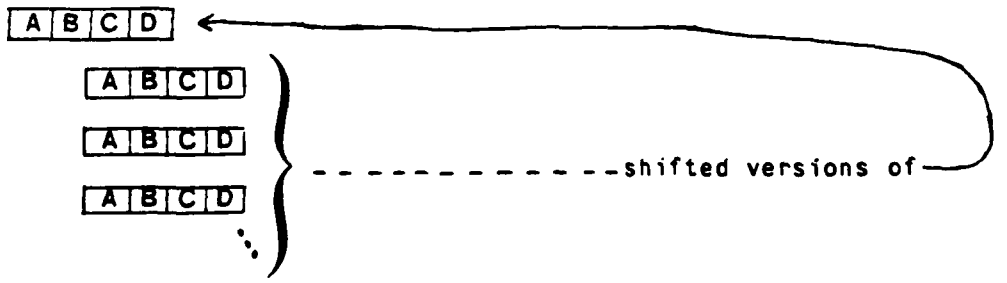
4.1.2.4.6 Shifting characters on a line

An example of filtered non-determinism is the specification of the SHIFT action, to support the user-command shift, causing characters on a (unit's) line to be shifted left or right. We define a relation ISSHIFTED between two sequences of characters and an integer, which holds if and only if the non-blank portion of the first sequence is equal to the non-blank portion of the second sequence, and has been shifted by the specified number of columns--by insertion of extra blanks if a positive number of columns, deletion of blanks if a negative number.

This we specify by

```
relation ISSHIFTED(NEWCHARS | sequence of character,
                   OLDCHARS | sequence of character, COLUMNS-TO-SHIFT | integer)
   definition
      ∃ old-left-blanks | sequence of " ", old-right-blanks | sequence of " ",
       new-left-blanks | sequence of " ", new-right-blanks | sequence of " ",
       non-blanks | sequence of character ||

          ~MEMBER(non-blanks , $ , " ") ∧
          oldchars = old-left-blanks @ non-blanks @ old-right-blanks ∧
          newchars = new-left-blanks @ non-blanks @ new-right-blanks ∧
          length(new-left-blanks) - length(old-left-blanks) = columns-to-shift ;
```

Observe that this definition implies that if there is at least one shifted version of a sequence (there might be none, if we tried to shift a non-blank character off the left of a sequence), then there exist an infinite number of equally valid shifted versions of the sequence (since extra blanks can always be tacked onto the end).



The sole purpose of this definition is to state what shifting means--the responsibility is left to the prevailing constraints of ensuring that the sequence of the appropriate length is inserted into a line of some unit-contents (namely, the one restricting character sequences of lines in units to be of length *Card-Length*). Thus, the shift action can merely select a sequence of characters which satisfies the ISSHIFTED relation to update each appropriate line. Notice that non-determinism has been used in two powerful ways here. First, through the ISSHIFTED relation, to select an appropriate object defined by its acceptance criteria without specifying how it should be computed. Second, the constraint on the length of lines, further filtering the satisfactory objects. The main difference between the two is that the first is a purely local acceptance criteria (as defined by the relation), while the second deals with the object usage (arbitrarily far into the future) and restricts choice to those which will not violate constraints.

The definition of action SHIFT using this is:

```
action  SHIFT[shift-command]
   definition
      over shift-command:LINES named line
      do update :CHARS of line to
                  some newchars | sequence of character ||
                     ISSHIFTED(newchars,chars,shift-command:COLUMNS) ;
```

Notation: some *x* | type || P(x) selects any object of the type that satisfies P(x).

4.1.2.4.7 The use of historical reference

Given that our modeling is based upon states and transitions between them, it is particularly convenient to have the ability to extract information from any previous state. If we did not have the power to make such "historical references", we would be forced to remember all information that might possibly be needed at some time in the future. This would both clutter up the information carried along from state to state and complicate our activities (since they would have to save such information explicitly). Thus historical reference is another example of a specification technique that allows us to limit our concerns to only those which are of current importance.

Once we have accepted the use of historical reference, it is natural to have the full power to refer to states as we have with any other type (with the limitation that we cannot change what has already happened!). Furthermore, observable events include not only the primitive transitions, but also the start and completion of action invocations.

4.1.2.4.8 Historical reference and inserting lines into units

One of the sub-commands of CHANGE is INSERT, to insert a sequence of lines into the unit-contents of a unit. The user optionally provides a specific location for the insertion to occur. If it is omitted, the lines are to be inserted after the last line changed in the unit since the start of the CHANGE command, or at the front of the unit if there has been no such change.

To model this we first define a relation between a unit and the latest state (since last starting CHANGE on that unit) in which the unit contents then differed from the unit contents now:

```
relation LATEST-STATE-OF-CHANGE( UNIT , CHANGESTATE | state)
   definition
      changestate = latest some state || unit: UNIT-CONTENTS ~equiv
                                         ( unit: UNIT-CONTENTS as of state ) ∧
                                   state after latest start CHANGE[unit,$] ;
```

Notation: <expression> as of <state> causes the expression to be evaluated in that state. latest <state> , where <state> is some expression denoting a state or states, denotes the most recent such state. start <action>[<object>,...] denotes the state(s) in which invocation of that action (with those objects as actual parameters) began. $state_1$ after $state_2$ is true if and only if $state_1$ is more recent than $state_2$.

Thus in the above "*state* after latest start CHANGE[*unit*,$]" will be true if and only if *state* is more recent than the latest start of CHANGE on *unit*.

Then, if such a state exists, we may determine the location of the change and insert the new lines after it; otherwise insert them at the front of the unit.

4.1.2.4.9 Historical reference and backup/restore commands

In the English specification of the source-data maintenance system, backup and restore commands cause portions of the world (libraries, files, etc.) to be written to or recovered from magnetic tape. In our modeling of the behavior (as opposed to implementation) within Gist, backup becomes a totally superfluous operation, since with historical reference we may refer to any past state and perform a restoration of an object to the condition it was in in that state. The ramifications of this for any implementation are that at any point in time the user might request restoration of any object to any earlier state! If, however, we choose to insist that the user's restore commands may only restore objects to the state they were in when explicitly mentioned in a backup command, the implementation need be prepared to make only such restorations, rather than arbitrary ones.

This is an interesting case, a non-obvious line between specification and implementation. We follow the more restrictive style of restoration, retaining the backup command to mark an object and a state by its invocation:

```
action  BACKUP[backup-command] ;
```

A restore command must refer to a state at which a backup command was invoked, and to an object that was the object, or within the object, given as the argument to the backup command:

```
action  RESTORE[restore-command]

   let obj : essence of project ∪ library ∪ file ∪ unit
         = restore-command:OBJECT

   precondition
      ∃ backup-command ||
            restore-command:backup-state = start BACKUP[backup-command] ∧
            ( backup-command:OBJECT contains obj )
                              as of restore-command:BACKUP-STATE

   definition
      restore obj as of restore-command:BACKUP-STATE ;
```

Notation: restore <object> as of <state> restores the object to the condition it was in in that state, creating/destroying objects and inserting/deleting relations to make the object equivalent (with respect to its type's boundary definition for equivalence--see 4.1.2.4.1) to its earlier condition. Thus restoring a file, for example, will involve deleting new units attributed to the file since the specified state and restoring the units that were attributed to the file in that state.

4.1.2.4.10 Historical reference and desired behavior

We have demonstrated how historical reference can be used to support BACKUP and RESTORE commands. It can also support the "temporary" change option. The user command CHANGE, to cause changes to a unit (with sub-commands INSERT, SHIFT, etc.), may be declared to be "temporary" in nature, in which case at the end of processing the current batch of user commands the (presumably) changed unit is to be restored to its state prior to commencement of the temporary changes.

We should, however, ask whether either form of restoration, either user-directed via the RESTORE command, or automatically because of earlier "temporary" changes, could possibly fail. It turns out that this possibility does indeed exist. Suppose we have two units, unit-A and unit-B, and unit-A

contains a line that "includes" unit-B. At this point we issue a backup command for unit-A, or change it temporarily. Now if we delete the "include" line in unit-A, we may then be able to change unit-B to insert a new line "including" unit-A. From this state it would be impossible to restore unit-A to its original condition, since this would violate our constraint prohibiting circularity of includes.

Should the attempted restoration simply fail, or should the modification of unit-B have been prevented? We must decide which of these (or other) alternative behaviors we want. The English specification, by not saying anything, suggests the first option, namely restoration can simply fail. Our feeling is that the possibility of such failure was never even recognized, and this behavior was dictated by circumstances rather than choice. Reasonable behavior might be that in the case of user-issued backup and restore commands, restore could fail in the manner described above (i.e., it is the responsibility of the user), whereas in the case of temporary changes, it must always remain possible to do the restoration (we don't want to surprise the user with an "automatic" facility that fails!). Our point is not that this is the "right" choice, rather that the existence of such a choice must be recognized and the choice consciously made, otherwise the resulting system may exhibit surprising and inappropriate behavior.

Our suggested choice permits us to demonstrate another Gist feature. We wish to write a constraint to ensure that restoration (of temporarily changed units) will always be possible. To do this we say

```
always required
   admissible RESTORE-TEMPORARILY-CHANGED-UNITS[] ;
```

where RESTORE-TEMPORARILY-CHANGED-UNITS is the action to perform the appropriate restoration.

The construct admissible <statement> is a predicate that is true if and only if there is some non-anomalous way of executing the statement in the current state of the world--however, the statement is not actually executed, i.e., the current state is not changed, but a truth value is returned. This is a simple form of "future" reference. Often the easiest way of predicting a result of doing something is to extract the desired information from the state in which it has been done, but proceed (with the information) from the state prior to doing it.

4.1.2.5 Dynamic constraints

The defined constraints serve to limit the possible states that the model world may achieve. We notionally distinguish between "static" and "dynamic" constraints by classifying those that refer only to the current state as "static", and those that refer to several states (usually the current state and the previous one) as "dynamic".

4.1.2.5.1 A "dynamic" constraint

An example of a dynamic constraint taken from the source-data maintenance world is:

" A unit may not be destroyed if it is included anywhere"

We express this constraint by forbidding a transition from a state in which a unit exists to a state in which that unit no longer exists (i.e., has been destroyed in the transition) but is still "included" by some line in the contents of a unit.

```
always prohibited
   ∃ dead-unit | essence of unit      ||
       destroyed dead-unit   ∧   UNIT-INCLUDES-UNIT($, dead-unit)
```

Reference to a past state is via the destroyed predicate, which is true if and only if the object existed in the immediately preceding state and has been destroyed in the transition.

This explicit statement of the constraint contrasts with the English specification, wherein no similar statement occurred; instead, an implementation to achieve the same effect (by making use of reference counts) was described. This was particularly inappropriate for several reasons: the intended behavior had to be deduced from the implementation; the other parts of code lost clarity because they had to deal with reference counts; worst of all, it was actually wrong - in the case of destroying an entire unit no mention was made of decrementing the reference counts of units "included" by lines of the destroyed unit.

4.1.3 Review of specification

The previous section demonstrated the use of Gist's features in some of the specification of the source-data maintenance task. The entire specification is available from the author.

We may now look back and ask the following questions:

1. How effective are the Gist features for expressing the structure and behavior of the task domain?

2. What have we learned about the task from having written a formal specification?

3. What difficulties arose in constructing the specification?

4. How comprehensible is the specification to someone other than the writer?

5. What do we intend to do with the formal specification?

4.1.4 Implications for Gist

Most of the Gist features used to specify the source-data maintenance task were present in the language prior to tackling this example. Certainly the underlying modeling approach proved satisfactory. We were led to some additions and notational refinements. These were:

- essence of, as a means of maintaining references to destroyed or yet to be created objects.
- boundary definitions, to support destroy, restore, etc.
- attribute notation for defining and making use of binary relations.

4.1.5 Implications for the source-data maintenance task

In constructing a formal specification in Gist we feel we have developed a much deeper understanding of the task than we had from merely reading the English description. We were led to recognition of ambiguities, implementation biases, and possible errors in the informal English. Formalizing in Gist forced us to consider behavioral interactions which we might otherwise have overlooked (for example, between restoration and constraints).

It may well be the case that spending a similar amount of time studying the task (without producing a Gist specification) would have led to the same insights, so we should regard the above observations as lack of negative evidence rather than additional positive support for our approach.

4.1.6 Difficulties of constructing the specification

A significant degree of effort was required to construct our specification. We attribute much of the difficulty to the need to infer the desired behavior from the English description. As we have already remarked, we found this description to be overly biased towards a particular implementation; hence we could not merely formalize the description, but rather had first to disentangle its intent and formalize that.

We were also hampered by a lack of experience of writing Gist specifications. Faced with a choice of alternative ways of specifying some behavior, we had little experience to guide our choice.

Finally, we recognize the need for some assistance in understanding the implications of our specification. In building a specification from descriptions of desired behaviors in simple cases we must be careful to consider the possible interactions of these cases and ensure that the resulting behavior is satisfactory.

4.1.7 Difficulties of understanding specification

We have observed that readers of our specification find it hard to understand--even other members of our group who have equal fluency in Gist. We attribute this in part to our emphasis on easing the task of the specification writer rather than the reader--Gist is still in development and there are undoubtedly many cosmetic improvements which would help. Nevertheless, there is more to specification than merely ensuring we have an effective notation. First, specifications may require some explanation of how they were derived--for example, make explicit the choices faced by the specifier and his reasons for choosing one over another. Second, some assistance is required to help the new reader build up gradually to a comprehension of the entire specification. We envisage several tools to assist comprehension. For example, a sophisticated interpreter would permit exploration of the behavior of a specification and analysis tools could outline the possible areas of interactions among the features of a specification.

Finally we might organize our specification into layers of increasing complexity, working from a simplified specification suppressing (or distorting) features in the higher layers, so that by descending from layer to layer an increasingly accurate and complete comprehension of the specification can be incrementally built up.

4.1.8 Use of specification

The main purpose of a Gist specification is to serve as a behavioral description. We must define which portions of its behavior are "observable" (e.g., observable behavior of the source-data maintenance specification would presumably include the creation / destruction of units, files, projects, etc., but not the invocation of particular actions PURGE, CHANGE. etc.). Given such a notion, we may ask whether an implementation exhibits the same observable behavior. If we can demonstrate that it does. we have validated that implementation. We have not as yet developed a syntax for expressing the observable portions of behavior of a Gist specification.

One of our long-term research aims is to be able to take formal specifications of behaviors written in Gist, together with relevant details suppressed from our behavioral description (namely, interface and implementation requirements, and refinements of the described normal-case behavior) and

transform them to achieve efficient implementations. We are sure that such transformations will require human guidance, since the distance between our specifications and any tolerable implementations is very wide.

For example, consider removing Gist constraints. Our aim would be to replace them by checks at all possible choice-points in the program that could lead to violation of the constraint. In cases where we cannot construct an appropriate predicate at the choice point to recognize which choices will lead to anomaly we would have to resort to other mechanisms, such as back-tracking. A detailed transformation which illustrates these problems is presented in [1].

4.2 HOST-IMP SPECIFICATION

4.2.1 Overall organization of specification

First we define general message-passing features common to imps and hosts, then we describe the specific features. We encompass the effects of user interaction and hardware failure as random activities with which the hosts and imps must be able to cope.

4.2.2 General message-passing features

We model message-passing by defining types **item** and **node**. Objects of type **node** transmit objects of type **item** between themselves. At this general level each item has two attributes--a MESSAGE (further details of which we will not need at this level) and an ADDRESS--the node to which the item is to ultimately be delivered. The Gist construct agent is used to describe nodes, in order that they may be independently active processes.

nodes - we denote connections between nodes via relation **CONNECTED**. For simplicity we prohibit a node from being connected to itself.
Each node has the following attributes:

- PENDING - those items yet to be dealt with by the node,
- TO-BE-SENT - items to be sent elsewhere by the node,
- MY-MESSAGES - messages extracted from items which are addressed to this node.

Each node has the following actions and demons:

- action **TRANSMIT** - this causes an item to be transmitted to a (connected) node.

- pending-demon **RECEIVE** - this demon is triggered when a transmission directed to this node is begun (i.e., a TRANSMIT action is invoked, with this node as the destination). Its response is to await completion of the TRANSMIT. If the completion was normal, then the 'transmitted' item is inserted into the node's pending attribute. (If the completion was abnormal, nothing is done, modeling the loss of a transmission if the transmitter fails during transmission).

- pending-demon PROCESS-PENDING - this demon is triggered when a new item is added to the node's pending attribute. Its response is to remove the item from the pending attribute (cleaning up), and on the basis of whether or not the item is addressed to this node, insert it into the my-messages attribute or the to-be-sent attribute.

- pending-demon **SEND** - this demon is triggered when a new item is added to the node's to-be-sent attribute. Since the response in this domain will turn out to be so dependent on whether the node is a host or imp, we will only present the definitions of SEND particular to hosts and imps.

- action ACTIVATE - this action is to be invoked when the agent is activated. It clears the to-be-sent and pending attributes, and waits *Relay-Set-Time* (a hardware requirement).

A separate agent **hardware failure** is defined to model the random hardware failure of nodes. Two demons are provided, one to (at random) deactivate a node, and the other to (re)activate a node. The intention is that this specify part of the environment in which the nodes operate, not that this be something the implementor must incorporate into his software!

4.2.3 Specialization to host-imp world

We refine the agent **node** into **host** and **imp**. We refine the type **message** into **user-message**, **acknowledgement** (to acknowledge successful receipt of an item), **host-dead-acknowledgement** (to acknowledge receipt of an item at the imp connected to the destination host when that host is currently "dead", and **host-going-down-message**, which a host sends its connected imp to indicate it is (voluntarily) going down. Since we are only concerned with host communication via imps, we prohibit CONNECT holding between two hosts.

A separate agent **interface** defines random demons to describe relevant possible behavior of the outside world. This consists of two demons,

- USER-SEND to create a user-message with destination some host and insert it into the to-be-sent attribute of a host;

- USER-TURN-OFF-HOST to create a host-going-down-message and insert it into the to-be-sent attribute of a host (this models the voluntary means by which a host goes down).

Agent host inherits the definitions of agent node. In addition, it also has the following:

- attribute CONNECTED-IMP, defined as the (unique) imp to which the host is CONNECTed (in both directions).

- pending-demon SEND - this demon is triggered by insertion of a new item into the host's TO-BE-SENT attribute. Its response is to TRANSMIT the item to its connected-imp (in point of fact, because of our constraints, the connected imp is the only node to which the host could possibly transmit the item).

Agent imp inherits the definitions of agent node and in addition has:

- Attribute DEAD-HOSTS, to record which of the hosts are known by this imp to be currently "dead" (down either due to hardware failure, or because they sent a host-going-down-message). This is maintained by two pending-demons:

  - MARK-HOST-DEAD, which inserts a (connected) host into the imp's dead-hosts when that host has become inactive or a host-going-down-message has been received by the imp from that host.

  - UNMARK-HOST-DEAD, which removes a host from the imp's DEAD-HOSTS attribute upon successful receipt of any item (other than one with a host-going-down-message as its message attribute) from that host (successful receipt is observed by watching for insertion of an item into the PENDING attribute).

- pending-demon SEND - this is triggered when a new item is inserted into the imp's TO-BE-SENT attribute. Its response is to first remove the item from that attribute; then it tests to see whether the imp is connected directly to the destination address of the item--if so, and if the destination is a host, then the appropriate acknowledgement must be sent and the item TRANSMITted (depending upon whether that host is currently regarded as dead). If the imp is not connected directly to the item's destination, then it is TRANSMITTED onwards through the network (in this specification we are not concerned with selecting which node in the network is the most appropriate to pass the item on to).

- pending-demon RECEIVE - a slight specialization of the RECEIVE defined for nodes, in that the response waits for termination of the TRANSMIT or 15 seconds (a "timeout"). As before, only normal completion of the TRANSMIT will cause insertion of the item into the PENDING attribute.

- pending-demon PROCESS-MY-MESSAGES, to discard messages addressed to this imp.

- action ACKNOWLEDGE-OK, used to create an acknowledgement for an item, addressed to the creator of that item.

- action ACKNOWLEDGE-DEAD, similar to ACKNOWLEDGE-OK, but used when the destination host is currently dead.

- demon CONNECTED-HOST-GONE-DOWN - this is triggered when a connected host is

observed to have become inactive. Its response is to invoke action **DEADIFY**, to discard all items addressed to that host at present in the imp's to-be-sent attribute, and to unschedule all pending SENDs of items to that host.

- demon **TARDY-SEND-QUEUES** is triggered if an item addressed to a connected host remains on the to-be-sent attribute more than 30 seconds. Its response is to invoke DEADIFY.

4.2.4 Implications for Gist

We required the following refinements of Gist:

- agents, which have independently active processes. In addition, such agents may be externally activated and deactivated. Deactivation is to abort the currently active and triggered processes of that agent. While deactivated, no demons of that agent will be triggered. Upon activating a previously deactivated demon, action **ACTIVATE** is invoked. None of the history prior to activation is available to that agent. Furthermore, agents must be able to voluntarily suspend themselves for some period of time via wait, during which time none of their demons are triggered. Following the delay, the currently active process continues from the point after the wait.

- Real time - "seconds" for wait delays, timeouts, etc.

- pending and non-pending demons. Non-pending demons are the usual Gist demons, which upon triggering are invoked immediately (perhaps interrupting the processing of another demon's response). For this specification we required another class of demons, which upon triggering would add themselves to the end of an implicit queue of already triggered demons. Only after the demons earlier on the queue had completed their responses would such a demon be permitted to commence processing its own response. This incorporation of a weak scheduling mechanism into the Gist demons was thought appropriate, rather than having to explicitly build a scheduler. We did some explicit program-controlled manipulation of the pending queue when some of the pending demons were to be unscheduled.

4.3 TEXT FORMATTER SPECIFICATION

4.3.1 Source of Problem

The formatter we specify is designed to have the capabilities of the formatter described by Kernighan and Plauger in chapter 5 of their book [2].

We give a (very) brief and informal account of the facilities the text formatter is to provide.

Input to the formatter is a sequence of lines, where lines consist of sequences of characters. Some lines will be text, some will be commands to the formatter. Command lines are identified by the

occurrence of a "." in the first column followed by a two letter abbreviation of the name of the command.

In action the formatter may be in a "fill" mode, during which paragraphs are formed by packing as many input words as possible into the output lines, the lines being "right-justified" (to produce an aligned right margin, like this paragraph) by padding out with extra spaces between words if necessary. When not in "fill" mode the input text lines are output without modification. With filling switched off, the words already gathered to go into the next output line are put out without right justification. This action of forcing out a partially collected line is called a break. Some of the commands implicitly cause breaks when they are encountered, even though they may not cause filling to be switched off.

We present the commands and briefly explain their actions:

	<u>"filling" commands</u>
fi	Cause a break and switch on "fill" mode.
nf	Switch off "fill" mode.
br	Cause a break (but does not switch into or out of "fill" mode).
	<u>page commands</u>
bp n	Begin page. n is an optional numeric argument, which, if present, is taken as the number of the new page. If not present the default is to increment the current page number by one. Causes a break. If this command would produce an entirely blank page (but for header and footer titles), i.e. occurs at the very top of a page, it merely adjusts the page number without creating the blank page.
pl n	Set page length to be n lines. Default is n = 66, does not cause break.
he t	Set the header to be printed at top of each page. t is a string argument which becomes the new header. The character " # " within the string is replaced by the current page number. Does not cause a break.
fo t	Set the footer title to be printed at bottom of each page. Analogous to the he command.
ls n	Set line spacing to n (i.e., n = 2 corresponds to double spacing). Default is n = 1, does not cause a break.
sp n	Causes a break and produces n blank lines. Default is n = 1. Does not produce blank lines at the very top of a page.

line commands

ce n	Cause a break and center the next n text lines (i.e., insert extra spaces if necessary to cause the text lines to be centered within the current margins.) Default is n = 1. If another ce command is encountered whilst centering text lines, the new command's value of n takes precedence.
ul n	Does not cause a break. Default is n = 1. As with ce command, encountering another ul command will adjust the count of lines to be underlined.
rm n	Set right margin to be n. Default is n = 60, does not cause a break.
in n	Set left margin (indentation) to be n. Default is n = 0, does not cause a break.
ti n	Cause a break and set the left margin for next output line only to be n. Default is n = 0.

Numeric arguments to commands may be preceded by a " + " or "-", in which case the value is taken to be the current value of the parameter being set incremented or decremented accordingly. An exception to this is the ti command which adjusts relative to the current left margin setting.

In order that the formatter behave reasonably with text containing a minimum of formatting commands, input lines which start with blanks or are entirely blank are treated as follows: Lines empty but for blanks cause a break and a blank line to be output (even at the top of a new page). Lines starting with n blanks (but followed by other characters) where n>0 cause a break and a temporary indent of + n.

This description includes many interface details of how the user is to indicate what formatting activities are to be applied to what portions of his text; we shall separate the specification of formatting activities from the specification of how to direct the application of these activities. Our specification is concerned with the former, namely the tasks of formatting, rather than the interface details. In Section 4.3.3 we briefly consider what must be done in an interface to our specified portion.

4.3.2 Formatter activities

Input is a sequence of

- "paragraphs" - each consisting of a sequence of "info←words", that is a word (sequence of characters) together with layout information (margins, page←size, etc.). The words are to be accumulated into right-justified lines.
- "info←lines" - a line (sequence of characters) together with layout information.

- "padding←lines" - these are to emerge as blank lines in the output, unless they would appear at the very top of a page, in which case they are to be discarded.

We follow the convention that the information associated with the first word to go into a line sets the characteristics for the entire line, and similarly that the information associated with the first line to go into a page sets the characteristics for the entire page.

Information relevant to lines consists of: left←margin, right←margin, line←spacing (the number of blank lines to be inserted between each text line) and subsequent←left←margin (if the word/line is too long to fit within the margins, it will be split over two or more lines--in such a case the left margin of subsequent lines is to be the subsequent←left←margin).

Information relevant to pages consists of header←title, footer←title, page←length and begin←page. This last indicates that a new page is to be started. In the original input we do not permit info←words other than the very first of a paragraph to cause a begin←page.

Output is a sequence of sequence of lines of characters, representing the page images produced as a result of the formatting operation.

4.3.3 Interface

The input as described by Kernighan and Plauger consists of a sequence of lines, with conventions for denoting text and formatting commands within this sequence. The input we assume pre-supposes some interface to derive from the user-input (in whatever form) the paragraphs, info←lines and padding←lines that we expect. To handle Kernighan and Plauger's input conventions our interface would have to

- recognize and decode commands;
- extract and accumulate words (to go into paragraphs) from text lines;
- process parameter setting commands to insert the appropriate information into info←lines and info←words (e.g. a command to set the right margin to some value would cause that value to be inserted into the inform- n of info←words/lines following that command until the next such command);
- convert space down commands into the appropriate number of padding lines.

4.3.4 Organization of specification

FORMAT is the top-level action invoked to do the formatting. This expects a sequence of paragraphs, info←lines and padding←lines which compose the input to be formatted.

Formatting is divided into two overall stages--first, the activities to form individual lines of output are performed, then pages are formed from these lines.

- FORMAT initiates the processing of the first stage by creating an object of type mixlets←input to which the input sequence is attributed. This will trigger the demons PARAGRAPHING, SPLIT←OVERLENGTH←LINES, CENTERING, LEFT←MARGIN←PADDING and INTER←LINE←PADDING to go to work on the input paragraphs and info←lines, and each perform their activity. Their net result is to convert the input sequence into a sequence of info←lines and padding←lines (no paragraphs left) ready to be incorporated into pages. There is some coordination between these demons to ensure that: overlength lines will be dealt with by SPLIT←OVERLENGTH←LINES before INTER←LINE←PADDING and LEFT←MARGIN←PADDING act on them, and CENTERING is performed on lines before LEFT←MARGIN←PADDING. Otherwise they act independently.

- After the first stage of formatting we are left with a sequence of info←lines and padding lines. Prior to forming pages, FORMAT converts this sequence into a sequence of simpler objects, which we call "linelets": Each linelet has two attributes, information, and either a line (of characters) or padding. Action PAGINATION is invoked on this sequence to produce the sequence of sequence of lines representing the page images.

We now briefly describe the activities of these demons and actions:

PARAGRAPHING replaces a paragraph (a sequence of info←words) by a corresponding sequence of (filled and right justified) info←lines. This is achieved by selecting a partition of the paragraph's words, and for each element of the partition (i.e., a sequence of words) forming an info←line.

- If the line is to represent just one word, no justification is done.

- If it represents two or more, it must be right justified (i.e., every word separated by at least one blank, and enough blanks inserted to align the leftmost and rightmost words with the left and right margins).

The only acceptable partitions are those for which such justification of the necessary lines is possible. Furthermore, we select the partition(s) which minimize the total number of info←lines produced (the essence of what a "filled" paragraph is).

SPLIT←OVERLENGTH←LINES - this demon splits overlength info←lines (i.e., info←lines for which the text is too wide to fit between the margins) into two (or more if necessary) info←lines. (Note that this also will take effect on lines produced during paragraphing, so excessively long words, i.e., so

long that the single word is too wide to fit between the margins--each of which will have been incorporated into a separate info←line, will be dealt with by this demon.)

CENTERING - this demon adjusts the left←margin value of those info←lines which are to be centered in order to center their text between the margins.

LEFT←MARGIN←PADDING - this demon inserts the appropriate number of blanks at the start of each text line which has a non-zero left←margin value.

INTER←LINE←PADDING - this demon inserts the appropriate number of padding lines after each text line when its spacing value is greater than zero.

PAGINATION - this action takes the sequence of linelets and forms a sequence of sequence of lines, the output page images. The process is similar to that applied to paragraphs; the incoming linelets are partitioned, and from each partition a page-image is constructed (involving discarding padding lines which would fall at the top of a page, adding/discarding padding lines at the bottom as necessary to bring the total page size to the appropriate length, including page header and footer if required). The acceptable partitions are those for which such construction is possible. From all acceptable partitions, we select the one(s) that leads to the least number of output pages.

4.3.5 Implications for Gist

4.3.5.1 New Gist usage

The primary novel use of Gist in this specification has been the construct continuations of ⟨statement⟩ from which ⟨object expression⟩ satisfies ⟨predicate⟩. This was applied within pagination and paragraphing, to do the appropriate activity in such a way as to minimize the resulting number of pages/lines. Hence we are applying a minimization predicate over all possible resulting states of the execution of ⟨statement⟩.

Extensive use was made of generator expressions to denote sequences and to iterate over sequences.

4.3.5.2 Exposed weaknesses of Gist and dissatisfaction with specification

On many occasions within this specification there is the need to perform some action which creates some new objects and communicates these back to the invoker--the invention and use of a unique relation for the sole purpose of such communication is a clumsy way of achieving this. Often this is

linked with the use of iterative constructs; there are powerful means of denoting expressions, and powerful means of iterating actions, but the combination of the two is not available (i.e., iterating actions and emerging with a result).

There seems to be a large overhead in creating all the syntactic support for relatively trivial actions, demons, and relations.

Could we perhaps have gone even further towards a descriptive rather than algorithmic specification? For example, the seemingly natural decomposition of the task into first forming lines and then combining these into pages might be regarded as an unfortunate split. It might compromise our ability to specify behaviors relying on the detailed interaction of these stages (e.g., padding out a filled and justified paragraph with extra blanks to make the overall paragraph length longer and so avoid "widows" or "orphans"--when the last line of a paragraph falls on the top of a new page, or the first line of a paragraph falls at the bottom of a page). Is there some suitable structuring of the specification that would not be biased in such a manner and yet be reasonably divided into comprehensible components?

4.3.5.3 Advantages accrued from the use of Gist

The increased ease of writing a specification rather than an algorithm is particularly evident in:

- The use of the continuations of... construct to free us first to describe a general means of performing some activity (e.g., forming the lines to represent a paragraph) and separately selecting which of the possibly multiple alternatives are appropriate (e.g., we want to minimize the resulting number of lines to represent that paragraph). Clearly we can always write a "British Museum" style algorithm to do the same, but Gist allows us to specify this effect with ease.

- The use of demons to direct the processing activities (e.g., centering, paragraphing, etc. are demons triggered by the data requiring the processing these demons perform) rather than having to write an arbitrary scheduler ourselves.

4.3.6 Extensions to formatter

Kernighan and Plauger suggest several possible extensions to their formatter's capabilities. Some of their extensions fall entirely within the portion we have chosen to regard as interface. We will limit our attention to those affecting the formatting activity itself. We consider how we might incorporate these extensions into our specification (which was not designed with these extensions in mind).

4.3.6.1 Separating justification and filling

We consider switching justification on and off separate from filling. A little consideration suggests that justification is meaningless outside of the context of paragraphs; hence we may limit our attention to the paragraphing activity.

We make use of a new relation TO←BE←JUSTIFIED, which holds of words in paragraphs whose lines are to be justified. As before, we presuppose that the interface has inserted this relation on the appropriate words.

Action CREATE←JUSTIFIED←INFO←LINES←FOR←PARAGRAPH requires modification to examine each sequence of words to go into a line and use CREATE←JUSTIFIED←INFO←LINE←FOR←WORDS or CREATE←UNJUSTIFIED←INFO←LINE←FOR←WORDS as appropriate (and following such modification might be better named CREATE←INFO←LINES←FOR←PARAGRAPH).

4.3.6.2 Filtering output pages

This extension allows us to limit our interest to a subset of the output pages. This may be easily incorporated by a modification to CREATE←PAGE←IMAGE, causing the empty sequence of lines to be "returned" if the page number is not among those in which we are interested.

4.3.6.3 Forcing text to appear on a single page

This extension is to permit us to force portions of the text to occur entirely in one page. To incorporate this extension, we must extend the datatype information to have an extra attribute, a "group"; the intended behavior is that all info←lines and info←words that are related to the same "group" must appear within the same page. In the formation of paragraph lines from words we must take care to ensure that the set of groups to which it related is the union of the groups to which its constituent words are related. The group information will be used by relation PAGE←PARTITION to require that info←lines related to the same group to fall within the same element of the partition.

4.3.6.4 Extra space after sentence

The purpose of this extension is to cause more than one space to follow the end of a sentence if it falls within the interior of a line. We may assume that this is applicable only during paragraph formation. The effect could be achieved by first defining an end-of-sentence recognizer; then within ARBITRARILY←PAD←WORDS←WITH←BLANKS we may examine all words except for the very last one to go on the line and, if it is recognized as a sentence terminator, require that it be followed by at least two blanks.

4.3.6.5 Hyphenation during filling

We will not consider the task of determining suitable hyphenation points of words; if we were somehow able to make such a determination we could use the information as follows:
Within CREATE←JUSTIFIED←INFO←LINES←FOR←PARAGRAPH we would extend the partitioning to include not only all possible partitions of (undivided) words but also partitions in which a single word could be split (according to our legal hyphenation scheme) over the end of one partition element and onto the start of the next. The characters of the first portion of the split word would be appended with the hyphenation character ("-"), and go to form the last info←word of the first partition element, the remaining characters going into an info←word at the start of the next partition element (with the possibility of further hyphenation, which might be necessary if this is a verrrrry long word...).

For example, here are some words to be partitioned for paragraphing.

Here are some
words to be partit-
ioned for para-
graphing.

4.3.6.6 Conditionals

This class of extensions is to permit the formatting actions to be dependent on the formatting conditions. This is the least specific of the extensions, and we can only speculate as to how amenable our specification might be to catering for whatever power might be desired.

We anticipate that our specification style would permit such features to be incorporated in a reasonably straightforward manner--we may specify the alternative actions as a non-deterministic choice at the appropriate place for the definition of such actions, and make the selection at the appropriate place for the testing of the conditions.

5. GIST INITIAL OPERATING CAPABILITY DESCRIPTION

5.1 INTRODUCTION

The Gist Initial Operating Capability (IOC) is the first step toward a package of tools to help specifiers create, test, and maintain Gist specifications. The IOC provides the following facilities:

- A Gist *editor* which enables the specifier to interactively create and modify the source text of Gist specifications. The editor ensures that the specification is syntactically correct at all times.
- A *pretty printer* which formats the text of a specification for enhanced legibility.
- A Gist *evaluator* capable of executing Gist expressions, predicates, and statements.
- A Gist *executive* which operates in the context of a "current" specification. The executive permits the specifier to create an initial execution state, specified in Gist, and interact with the evaluator to explore possible activity from that state.
- The ability to save Gist specifications created in the IOC environment on files, and to load such saved specifications into the IOC.

The structure of the IOC is depicted in Figure 5-1. The entire IOC is implemented in Interlisp [3] and runs within the standard Interlisp environment. The user communicates through the Gist executive to the various Gist tools, and need not interact with or understand Interlisp at all. However, any command not specifically recognized as meaningful to one of the tools is treated as an Interlisp command. This means that the user has available the full facilities of Interlisp, including the Programmer's Assistant facilities. It also means that error messages in response to illegal inputs may appear obscure to users unfamiliar with Interlisp.

The editor and prettyprinter subsystems are constructed automatically (as Interlisp programs) by another program, called POPART, from a BNF syntax for Gist. These tools provide all access to the specification, both in its internal *tree* representation and its external *text* representation. The specification is tested by the Gist evaluator which is composed of a declaration compiler and an interpreter. The evaluator, through the interpreter, produces the behavior of the specification on the test data. The evaluator uses AP3, a programming language embedded in Interlisp, to maintain an internal representation of one or more chains of process states as the specifier tests his specification. Finally, the Gist executive interfaces the specifier to these tools.

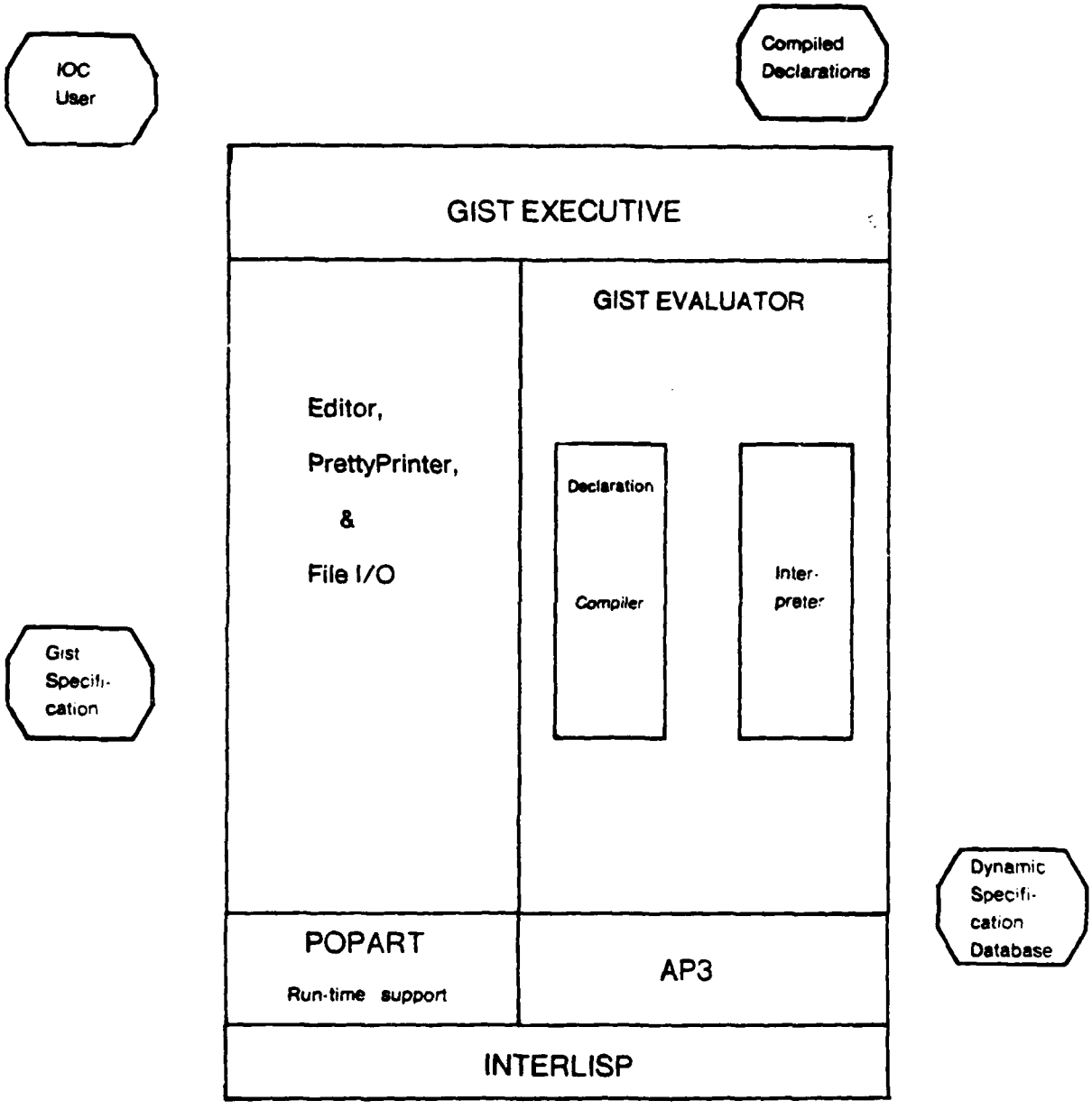


Figure 5-1: Gist IOC configuration

5.1.1 Transliteration

In much of the printed documentation on Gist, *including examples of Gist specifications*, symbols are used which are not available in the standard ASCII character set. The IOC requires that these symbols be transliterated as shown in Table 5-1.

Table 5-1: IOC symbol transliteration

Publication Symbol	IOC transliteration
$\wedge$	and
$\vee$	or
$\otimes$	xor
$\Rightarrow$	implies
$\equiv$	equiv
$\neg$	not , ~
$\forall$	for all, all
$\exists$	there exists, exists
$\in$	element of
$\cup$	union
$\cap$	intersect

5.2 Editor, PrettyPrinter, and File I/O

The Gist editor, parser, and prettyprinter are produced automatically from a BNF definition of Gist's syntax by a program named POPART. The full complement of commands and capabilities afforded by the resulting tools is documented in [5]. A subset of these commands, sufficient for convenient creation and editing of Gist specifications, is documented below. Each command has a short name, mnemonic for the function it performs. Some commands also require an operand. A command is invoked simply by typing its name (and operand) in response to the prompt "←" from the IOC executive. (Each prompt is preceded by a number, which can be used in Programmer's Assistant commands to refer to the corresponding IOC event.) If the command needs no operand, its name should be followed immediately by a carriage return, without intervening spaces. If the command needs an operand which is not Gist text, the command name should be followed by a space, and then by the operand and a terminating carriage return. If the operand is Gist text, the command should be followed by a carriage return, then the Gist text (which may be spread across as many lines if convenient) terminated by two periods and a carriage return.

The editor maintains at all times a specification being edited and a current focus (syntactic constituent) within that specification. Initially, both are empty. The editing commands will be described in two groups:

- commands which alter the specification
- commands which change the editor's focus within the specification

5.2.1 Commands which Alter Specification Text

```
←Set
Gist text ..
```

The Set command requires a single operand, a Gist text string. That text becomes the current specification and the current focus.

```
←Replace
Gist text ..
```

The Replace command requires a single operand, a Gist text string. That text replaces the current focus within the specification, and becomes the new focus.

The Replace command is also used to *embed* existing text in larger text. This is done by using the special string "$$" to refer to the existing text within the embedding text. For instance, to replace the Gist text "ship:captain" with "ship:captain:salary" either of the following commands would suffice:

```
←Replace
ship:captain:salary ..

←Replace
$$:salary ..

←ReplaceAll
Gist text ..  ==>  Gist text ..
```

The ReplaceAll command replaces every occurrence of the first text fragment, which appears as a syntactic constituent of the current focus, with the second text fragment. The current focus is not changed. This command is most useful for changing some identifier name in a specification.

```
←After
Gist text ..
```

The After command adds its operand after the current focus. The current focus must be an element of an "iterated field" in the grammar. In Gist, this means a statement or declaration within a role specification within the list of roles of an action or relation declaration, etc. In general, if the constituent being added is the same "kind" of thing as the current focus constituent, the After command is probably allowed. So, for example, a type declaration can be added after any other using the After command. If, on the other hand, the current focus combines syntactically

with the new text to form a new constituent of the same "kind" as the current focus, the **Replace** command should be used. So, to change a specification which read "officer:salary" to read "officer:salary * 1.1" the **Replace** command, rather than the **After** command, is appropriate. The **After** command does not change the current focus.

←Before
Gist text ..

The **Before** command is like the **After** command, but places the new text immediately before the current focus. within an iterated field.

←Delete

The **Delete** command deletes the current focus, which must be an element of an iterated field or an optional element. When deleting an element of an iterated field other than the final element, the separator following the element is also deleted. For instance, deleting a statement in a block will delete the ";" following it as well.

No change to a specification is ever permitted if that change would result in a syntactically incorrect specification. An attempt to make such a change will produce an error message indicating a point in the text beyond which no attempted parse could progress. The specification and current focus are left unchanged by the attempt.

5.2.2 Commands which Change the Editor's Focus

←Find
Gist Text..

The text must be a complete syntactic constituent. Thus "salary" and "salary * 1000" are acceptable, but "salary *" is not. The **Find** command looks for an identical constituent within or beyond the current focus. The first one found becomes the new focus. If none is found, the focus is not changed.

←Top

The **Top** command changes the current focus to be the entire specification.

←In

If the current focus is an "iterated field" e.g., the statements of a block, the **In** command resets the focus to the first element of the iterated field. Otherwise, the current focus is set to some immediate subexpression of the focus.

←Out

The **Out** command resets the focus to its immediate parent constituent.

Next integer

If the current focus is an element of an iterated field, the **Next** command can be used to change the focus to another element of the field. The desired element is designated by an integer operand. The positive integers designate has current focus as 1 and its successors as 2,3,.... The negative integers designate the elements from the last (-1) back to the first. If the operand is omitted, the focus is reset to its immediate successor in the iterated field.

←Previous

If the current focus is an element of a iterated field, **Previous** resets the focus to its predecessor in that field.

5.2.3 PrettyPrinting

The command **Pretty** may be used at any time to display a formatted version of the current editor focus. The Gist prettyprinter automatically breaks the text into lines, inserting spacing and indentation appropriately to enhance readability of the text. Examples of the format produced may be seen in Sections 6.1 and 6.2.

5.2.4 Commands for Saving and Restoring Specifications

←Read <filename>

The **Read** command expects a single parameter, a Tops-20 filename. **Read** restores the specification saved on the named file. The specification and current focus in effect are both replaced by the specification read from the file. This file should be one created by the **Write** command. The **Read** command is equivalent to using the editor's **Set** command and retyping the specification.

←Write <filename>

Write makes a prettyprinted copy of the current specification on the named file.

5.3 Specification Testing

In order to test a specification, the specifier must have

- a Gist specification
- an "initial state" from which testing is carried out
- some behavior to be tested

A Gist specification consists of a block of declarations, bracketed by **begin** and **end**. The IOC expects these declarations to be augmented by one or more Gist statements placed, by convention, in this outermost block of the specification following the declarations. It is the function of these statements to establish an initial state from which testing is carried out. The actual test behavior is then run by interaction with the Gist evaluator, which consists of interdependent compiler and interpreter portions.

5.3.1 The Gist Declaration Compiler

In order to permit the declarations (of types, relations, etc.) to affect the evaluation of the commands used to test the specification, the IOC performs a compilation of the declarations appearing in the specification. This compilation process is initiated by the command **gist**:

```
←gist
```

In response to this command, the IOC compiles the declarations and places the user in an environment for interaction with the Gist evaluator. In this environment, interactions are prompted by the symbol "->", and numbered in a sequence independent from the numbering of interactions with the editor. To resume editing operations, the user should type "OK" in response to the "->" prompt.

5.3.2 The Gist Evaluator

The IOC permits the user to provide Gist statements, predicates, and expressions interactively for evaluation relative to the "current process state". An initial state is established by the **init** command:

```
->init
```

This command causes the evaluator to execute all statements in the specification's outermost block. These statements are executed sequentially in order of appearance, and *without* the semantic effects of any constraints in the specification, (including type requirements). It is currently the user's

responsibility to ensure that the resulting state is a valid one.[15] Typically, the initial state is established by means of a single atomic statement, which creates an adequate process environment for the desired testing.

Interactive testing relative to the current state is ordered by the ".." command:

```
->..
object expression, predicate, or statement ..
```

The text of the Gist constituent to be evaluated may extend across several lines of terminal input. The final line must terminate with "..". Provided the input parses as one of the three syntactic classes indicated, it is processed, according to its syntactic class, as follows:

- *Object expressions* are evaluated relative to the current state. If the expression has no referent, "Failed..." will be printed. Otherwise, some referent of the expression will be displayed. The current process state does not change.

- *Predicates* are also evaluated relative to the current state. The predicate's value, TRUE or FALSE, is displayed. If the value is TRUE, and the predicate has leading existentially quantified variables, a set of bindings for those variables is also displayed. Conversely, if the value is FALSE and the predicate has leading universally quantified variables, a set of bindings for these variables is displayed. The current process state does not change.

- *Statements* are executed from the current state. If the statement has any valid executions some valid execution is carried out, and "success" is displayed. The final state of this execution becomes the current process state, and is linked via the states traversed in attaining it back to the previous current state. Otherwise "Failed..." is printed and the current state remains unchanged. In this case the IOC command "**why**"

  ```
  ->why
  ```

  will provide some indication of a reason for failure of the statement.[16]

The Gist constituent to be evaluated by the ".." command may make use of any types, relations, attributes, and actions defined in the user's current specification.

Since a Gist process state is independent of variable names, each interaction takes place in a bare variable environment - variables may not be used freely in these inputs. They may of course establish their own internal binding environments.

---

[15] Gist's semantics define legitimate transitions from valid states to new valid states. Until an initial valid state is established, the constraints cannot be enforced.

[16] This reason is a constraint violated along the "longest" partial execution path, as measured by a heuristic distance metric.

5.3.3 Literal and Created Objects

Integer and character sequence constants may be included in specifications and IOC interactions by use of Interlisp integer and string constants.[17] Constants of other base types are included by using as an expression an identifier whose name does not contain a "." and which is not used as a variable name in the lexical context in which it appears. All such uses of a given identifier refer to some object, having that identifier as its "internal" name. All other objects are created by the execution of a Gist create statement, and are given internal names consisting of the first few characters of the objects type and several digits.[18]

5.3.4 Input/Output Facilities

Although there are no defined I/O activities in Gist, the IOC provides a few basic facilities for testing interactive parts of specifications.

Terminal input to a process may be provided by the pseudo expression GISTREAD. GISTREAD (prompt-string) may appear as an object expression anywhere in a specification. The evaluator will process this expression by printing the referent of prompt-string, which may be any Gist object expression but is typically a string constant. The evaluator will then read a single Gist object expression (typically a literal) from the terminal and its referent becomes the referent of the GISTREAD expression.[19]

Terminal output is provided by the pseudo action GISTPRINT. GISTPRINT(object expression) may appear as a statement anywhere in a specification. The referent of the object expression is output to the terminal. The output consists of the internal name of the object expression's referent, together with its "boundary" as specified in any boundary declaration for the referent's type. The Gist syntax for boundary declarations is extended in the IOC to permit specification of a print boundary. For example, the declaration:

bound employee for print by :salary, ::supervisor

would cause GISTPRINT, when printing an object of type employee, to print not only the object's internal name but to display its salary and all other objects having it as their supervisor. In displaying

---

[17] Real numbers are not currently supported by the IOC.

[18] The IOC test for object identity is based on these internal names. For this reason, literals should NOT be given names consisting of characters followed by digits.

[19] GISTREAD may be considered to be a facility which allows the user to simulate an unspecified portion of a system.

the boundary, the values are themselves printed according to any print boundaries declared for their types. *This recursive printing of boundaries is cut off at a depth controlled by the user. The IOC* command

```
-> GISTPRINTDEPTH [integer]
```

resets the cutoff depth to the given integer. A depth of 0 indicates printing of an object's internal name only. Depth N indicates printing an object's internal name, together with printing its print boundary to a depth of N-1. The initial depth setting is 0.

Since all terminal I/O, whether direct commands to the IOC or GISTREAD or GISTPRINT interactions. is performed through Interlisp, transcripts of sessions may be made with Interlisp's DRIBBLE facility. Terminal characteristics, such as linelength, may be established by the conventional Interlisp mechanisms.

5.4 Debugging/Testing Aids

A primitive form of debugging can be obtained by interspersing GISTPRINT statements at strategic points in a specification. Another facility is the ability to "trace" or "break" certain activity.

Tracing consists of printing information on the terminal in response to various state changes. Breaking is similar to tracing. but also provides a pause in execution during which the user may interact with the IOC. In particular. he may evaluate predicates and expressions relative to the statement at which the break occurs. When satisfied, he gives an "OK" command and processing continues.

Tracing and breaking are turned on by the IOC commands GTRACE and GBREAK:

```
->GTRACE [name1, ..., namen]
->GBREAK [name1, ..., namen]
```

where each $name_i$ may be:

- An action name, in which case invocations of the action are traced/broken. The action name. together with the names of its formal parameters and corresponding actual parameters are printed.
- A relation name, in which case every insertion or deletion of a relationship in the relation is traced/broken.
- An attribute name, in which case each change involving that attribute is traced/broken.
- A type name, in which case classification and de-classification of objects in the type is traced/broken (including creation and destruction of objects in the type).

The command forms:

```
->GTRACE [T]
->GBREAK [T]
```

can be used to trace/break all activity which may be traced/broken.

The user must be aware that in evaluating a statement, the evaluator is conducting a search for a valid execution. Tracing and breaking takes place as the search progresses. Some of the broken or traced activity may be taking place along paths of the search which will be abandoned, not leading to valid executions.

Tracing/breaking may be turned off selectively by the commands:

```
->GUNTRACE [name1, ..., namen]
->GUNBREAK [name1, ..., namen]
```

T can be used as an operand to these commands to turn off all enabled tracing/breaking, regardless of whether it was enabled with the "T" operand.

5.4.1 Modifying a Specification

After creating and testing a specification, the user will likely want to make changes to it. This is done by exiting the Gist evaluator (via the "OK" command) and giving the editor commands necessary to make the desired change. Most changes to the specification will require re-analysis of part(s) of the specification for the changes to affect further testing. Currently this is only possible by reentering the Gist evaluator (via the "gist" command) and beginning further testing with the "init" command, which re-establishes the "initial" state as the "current" state.

5.5 IOC Coverage

Gist is still undergoing frequent change, both to the syntax of established language constructs and by additions to the language. Appendix I contains the Gist grammar used by the IOC as of October, 1980. This is a context free grammar in the notation used by the POPART system. The Gist editor may be used to create any specification which is a "statement" in this grammar.

The IOC evaluator is not capable of evaluating this full range of specifications. Among the important Gist concepts not covered by the evaluator are:

- temporal reference (with a few minor exceptions)
- sets and sequences (multivalued attributes are handled, however)

- orderings (except for the ordering of integers)
- generators
- demons
- pseudo-parallelism

In Gist, literal objects are normally introduced in the declarations of their types. The IOC declaration compiler does not yet process these declarations. It is therefore necessary to establish the type of any non-numeric literals used in the specification by means of the statement(s) setting up the initial state. This is accomplished by the inclusion of statements of the form:

insert literal isa type

6. IOC TRACES

6.1 IOC TRACE--PSL EXAMPLE

Below is a trace of a user interacting with the IOC to explore behavior of a simplified specification of the Program Support Library.

```
88-Read PSL.GST

89-Pretty
begin
  type project(any library::unique);
  type library(any file::unique);
  type file(any unit::unique);
  type unit(any unit-contents:line)bound destroy
        by equiv:unit-contents;
  type line(any chars:character,any includes-unit:unit);
  type character;

  relation
    UNIT-INCLUDES(includor:unit,included:unit)
  definition
    there exists line|includor:unit-contents=line and
                      (line:includes-unit=included or
                       UNIT-INCLUDES(line:includes-unit,included));

  always prohibited CIRCULAR-INCLUSION there exists unit|
  UNIT-INCLUDES(unit,unit);
  type update-command(current-file:file,any update-sub-commands:
                          update-sub-command);
  type update-sub-command(optional current-unit:unit)
        supertype of<add-command;
                     purge-command;
                     change-command>;

  type add-command(any add-data:line);
  type purge-command;
  type change-command(any change-sub-commands:change-sub-command);
  type change-sub-command supertype of<insert-command;
                                       modify-command;
                                       delete-command>;
  type insert-command(any insert-data:line);
  type modify-command(optional line-to-mod:line,oldchar:character,
                      newchar:character);
  type delete-command(any line-to-del:line);
```

```
action
  UPDATE[update-command]let file=update-command:current-file,
up-com:update-sub-command=update-command:update-sub-commands
definition
  case up-com of add-command
    =>ADD[file,up-com];
  purge-command
    =>PURGE[up-com:current-unit];
  change-command
    =>CHANGE[up-com:current-unit,up-com];
end case;

action
  ADD[file,add-command]
definition
  atomic create unit.1;
         insert all unit.1:unit-contents=add-command:add-data;
         insert file:unit=unit.1
  end atomic;

action
  PURGE[unit]
    precondition not there exists including-unit:unit,
                              line|including-unit:unit-contents=line
    and line:includes-unit=unit
definition
  destroy unit;

action
  CHANGE[unit,change-command]let ch-com:change-sub-command=
    change-command:change-sub-commands
definition
  case ch-com of insert-command
    =>INSERT[unit,ch-com];
  modify-command
    =>MODIFY[unit,ch-com];
  delete-command
    =>DELETE[unit,ch-com];
end case;

action
  INSERT[unit,insert-command]
definition
  insert all unit:unit-contents=insert-command:insert-data;
action
  DELETE[unit,delete-command]
definition
  delete all unit:unit-contents=delete-command:line-to-del;
```

```
action
  MODIFY[unit,modify-command]let oldchar:character=modify-command:
    oldchar,newchar:character=modify-command:newchar
definition
  if there exists line|modify-command:line-to-mod=line
       then DO-MODIFY[line,unit,oldchar,newchar]
       else DO-MODIFY[(a line|DO-MODIFY(line,unit,
                                        $,$)asof ever),
                      unit,oldchar,newchar];

action
  DO-MODIFY[line,unit,character.1,character.2]
    precondition line=unit:unit-contents
definition
  if line:chars=character.1
       then begin
              delete line:chars=character.1;
              insert line:chars=character.2
            end;

atomic insert myproject isa project;
       insert mylibrary isa library;
       insert myfile isa file;
       insert myproject:library=mylibrary;
       insert mylibrary:file=myfile;
       insert chara isa character;
       insert charb isa character;
       insert charc isa character;
       insert chard isa character;
       insert chare isa character;
       insert charz isa character;
       insert unita isa unit;
       insert myfile:unit=unita;
       insert unitb isa unit;
       insert myfile:unit=unitb;
       insert unitc isa unit;
       insert myfile:unit=unitc;
       insert linea isa line;
       insert linea:chars=chara;
       insert linea:includes-unit=unitb;
       insert unita:unit-contents=linea;
       insert lineb isa line;
       insert lineb:chars=charb;
       insert unitb:unit-contents=lineb;
       insert linecd isa line;
       insert linecd:chars=charc;
       insert linecd:chars=chard;
       insert linecd:includes-unit=unita;
       insert unitc:unit-contents=linecd;
       insert linee isa line;
       insert linee:chars=chare;
```

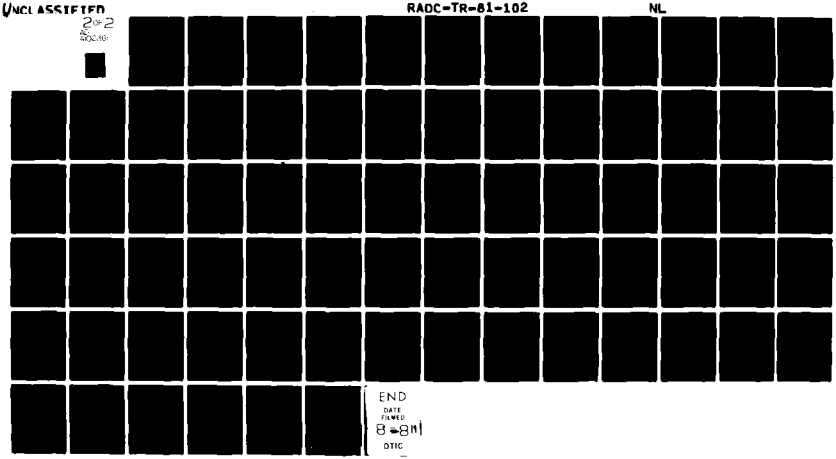
```
insert purge1 isa purge-command;
insert purge1:current-unit=unitb;
insert update1 isa update-command;
insert update1:current-file=myfile;
insert update1:update-sub-commands=purge1;
insert delete2 isa delete-command;
insert delete2:line-to-del=linea;
insert change2 isa change-command;
insert change2:change-sub-commands=delete2;
insert change2:current-unit=unita;
insert update2 isa update-command;
insert update2:update-sub-commands=change2;
insert update2:current-file=myfile;

insert line3 isa line;
insert line3:includes-unit=unitc;
insert insert3 isa insert-command;
insert insert3:insert-data=line3;
insert change3 isa change-command;
insert change3:change-sub-commands=insert3;
insert change3:current-unit=unita;
insert update3 isa update-command;
insert update3:update-sub-commands=change3;
insert update3:current-file=myfile;

insert modify4 isa modify-command;
insert modify4:line-to-mod=linecd;
insert modify4:oldchar=charc;
insert modify4:newchar=charz;
insert change4 isa change-command;
insert change4:change-sub-commands=modify4;
insert change4:current-unit=unitc;
insert update4 isa update-command;
insert update4:update-sub-commands=change4;
insert update4:current-file=myfile;

insert insert5 isa insert-command;
insert insert5:insert-data=linee;
insert change5 isa change-command;
insert change5:change-sub-commands=insert5;
insert change5:current-unit=unitc;
insert update5 isa update-command;
insert update5:update-sub-commands=change5;
insert update5:current-file=myfile;
```

```
        insert modify6 isa modify-command;
        insert modify6:oldchar=chard;
        insert modify6:newchar=chara;
        insert change6 isa change-command;
        insert change6:change-sub-commands=modify6;
        insert change6:current-unit=unitc;
        insert update6 isa update-command;
        insert update6:update-sub-commands=change6;
        insert update6:current-file=myfile
  end atomic
end..

90-gist

              **** G I S T ****

type "help" for command list
(re)processing declarations
3->GTRACE(T]
T
4->init
inserting myproject isa project in cx:2
inserting mylibrary isa library in cx:2
inserting myfile isa file in cx:2
inserting myproject : library = mylibrary in cx:2
inserting mylibrary : file = myfile in cx:2
inserting chara isa character in cx:2
inserting charb isa character in cx:2
inserting charc isa character in cx:2
inserting chard isa character in cx:2
inserting chare isa character in cx:2
inserting charz isa character in cx:2
inserting unita isa unit in cx:2
inserting myfile : unit = unita in cx:2
inserting unitb isa unit in cx:2
inserting myfile : unit = unitb in cx:2
inserting unitc isa unit in cx:2
inserting myfile : unit = unitc in cx:2
inserting linea isa line in cx:2
inserting linea : chars = chara in cx:2
inserting linea : includes-unit = unitb in cx:2
inserting unita : unit-contents = linea in cx:2
inserting lineb isa line in cx:2
inserting lineb : chars = charb in cx:2
inserting unitb : unit-contents = lineb in cx:2
inserting linecd isa line in cx:2
inserting linecd : chars = charc in cx:2
inserting linecd : chars = chard in cx:2
inserting linecd : includes-unit = unita in cx:2
inserting unitc : unit-contents = linecd in cx:2
inserting linee isa line in cx:2
inserting linee : chars = chare in cx:2
inserting purge1 isa purge-command in cx:2
```

AD-A102 361 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
DESIGN SPECIFICATION VALIDATION.(U)
JUN 81 R M BLAZER F30602-79-C-0042

UNCLASSIFIED RADC-TR-81-102 NL

2 OF 2
AD A102 361



END
DATE FILMED
8-81
DTIC

```
inserting purge1 : current-unit = unitb in cx:2
inserting update1 isa update-command in cx:2
inserting update1 : current-file = myfile in cx:2
inserting update1 : update-sub-commands = purge1 in cx:2
inserting delete2 isa delete-command in cx:2
inserting delete2 : line-to-del = linea in cx:2
inserting change2 isa change-command in cx:2
inserting change2 : change-sub-commands = delete2 in cx:2
inserting change2 : current-unit = unita in cx:2
inserting update2 isa update-command in cx:2
inserting update2 : update-sub-commands = change2 in cx:2
inserting update2 : current-file = myfile in cx:2
inserting line3 isa line in cx:2
inserting line3 : includes-unit = unitc in cx:2
inserting insert3 isa insert-command in cx:2
inserting insert3 : insert-data = line3 in cx:2
inserting change3 isa change-command in cx:2
inserting change3 : change-sub-commands = insert3 in cx:2
inserting change3 : current-unit = unita in cx:2
inserting update3 isa update-command in cx:2
inserting update3 : update-sub-commands = change3 in cx:2
inserting update3 : current-file = myfile in cx:2
inserting modify4 isa modify-command in cx:2
inserting modify4 : line-to-mod = linecd in cx:2
inserting modify4 : oldchar = charc in cx:2
inserting modify4 : newchar = charz in cx:2
inserting change4 isa change-command in cx:2
inserting change4 : change-sub-commands = modify4 in cx:2
inserting change4 : current-unit = unitc in cx:2
inserting update4 isa update-command in cx:2
inserting update4 : update-sub-commands = change4 in cx:2
inserting update4 : current-file = myfile in cx:2
inserting insert5 isa insert-command in cx:2
inserting insert5 : insert-data = linee in cx:2
inserting change5 isa change-command in cx:2
inserting change5 : change-sub-commands = insert5 in cx:2
inserting change5 : current-unit = unitc in cx:2
inserting update5 isa update-command in cx:2
inserting update5 : update-sub-commands = change5 in cx:2
inserting update5 : current-file = myfile in cx:2
inserting modify6 isa modify-command in cx:2
inserting modify6 : oldchar = chard in cx:2
inserting modify6 : newchar = chara in cx:2
inserting change6 isa change-command in cx:2
inserting change6 : change-sub-commands = modify6 in cx:2
inserting change6 : current-unit = unitc in cx:2
inserting update6 isa update-command in cx:2
inserting update6 : update-sub-commands = change6 in cx:2
inserting update6 : current-file = myfile in cx:2

5->..
charb::chars..
lineb
6->..
lineb:chars..
```

```
charb
7->..
linecd:chars..
charc
8->redo 7
chard
9->redo 7
charc
10->redo 7
chard
11->redo 7
charc
12->redo 7
chard
13->..
exists file | file:unit:unit-contents = linea..
(TRUE ((file . myfile)))
14->..
linee::unit-contents..
Failed...
15->..
UPDATE[update1]..

UPDATE:
update-command = update1

PURGE:
unit = unitb
Failed...
16->why

failing precondition:
not there exists including-unit:unit,line | including-unit:unit-contents=
    line and line:includes-unit=unit..

Variable bindings:
((unit . unitb))

17->..
PURGE[unita]..

PURGE:
unit = unita
Failed...
18->why

failing precondition:
not there exists including-unit:unit,line | including-unit:unit-contents=
    line and line:includes-unit=unit..

Variable bindings:
((unit . unita))

19->..
UPDATE[update2]..
```

```
UPDATE:
update-command = update2

CHANGE:
unit = unita
change-command = change2

DELETE:
unit = unita
delete-command = delete2
deleting unita : unit-contents = linea in cx:7
DELETE completed
CHANGE completed
UPDATE completed
success
20->..
unita:unit-contents..
Failed...
21->..
destroy myfile..
deleting myfile isa file in cx:11
deleting mylibrary : file = myfile in cx:11
deleting myfile : unit = unitc in cx:11
deleting myfile : unit = unitb in cx:11
deleting myfile : unit = unita in cx:11
deleting update6 : current-file = myfile in cx:11
deleting update5 : current-file = myfile in cx:11
deleting update4 : current-file = myfile in cx:11
deleting update3 : current-file = myfile in cx:11
deleting update2 : current-file = myfile in cx:11
deleting update1 : current-file = myfile in cx:11
Failed...
22->why

count spec constraint violation:
((current-fileofupdate-command myfile update1)
 (1 . 1)
 : 0)

23->?? 10 thru 20


10.  redo 7
     ->.. linecd : chars
     chard
11.  redo 7
     ->.. linecd : chars
     charc
12.  redo 7
     ->.. linecd : chars
     chard
13.  ->.. exists file | file : unit : unit-contents = linea
     (TRUE ((file . myfile)))
14.  ->.. linee :: unit-contents
```

```
     Failed...
15.  ->.. UPDATE %[ update1 %]
     Failed...
16.  ->why

17.  ->.. PURGE %[ unita %]
     Failed...
18.  ->why

19.  ->.. UPDATE %[ update2 %]
     success
20.  ->.. unita : unit-contents
     Failed...


23->..
UPDATE[update1]..

UPDATE:
update-command = update1

PURGE:
unit = unitb
deleting unitb isa unit in cx:14
deleting myfile : unit = unitb in cx:14
deleting linea : includes-unit = unitb in cx:14
deleting unitb : unit-contents = lineb in cx:14
deleting purge1 : current-unit = unitb in cx:14
deleting lineb isa line in cx:14
deleting lineb : chars = charb in cx:14
deleting unitb : unit-contents = lineb in cx:14
PURGE completed
UPDATE completed
success
24->..
insert lineg isa line..
inserting lineg isa line in cx:17
success
25->..
insert lineg:chars = charb..
inserting lineg : chars = charb in cx:18
success
26->..
insert unitg isa unit..
inserting unitg isa unit in cx:19
Failed...
27->why

count spec constraint violation:
((unitoffile unitg unitg)
 (1 . 1)
 :: 0)

28->..
atomic
```

```
   insert unitg isa unit;
   insert unitg::unit = myfile
end atomic..
inserting unitg isa unit in cx:20
inserting myfile : unit = unitg in cx:20
success
29->..
insert unitg:unit-contents = lineg..
inserting unitg : unit-contents = lineg in cx:21
success
30->..
exists unit | UNIT-INCLUDES(unit,unitg)..
(FALSE)
31->..
insert linea:includes-unit = unitg..
inserting linea : includes-unit = unitg in cx:22
success
32->..
insert unita:unit-contents = linea..
inserting unita : unit-contents = linea in cx:23
success
33->..
exists unit | UNIT-INCLUDES(unit,unitg)..
(TRUE ((unit . unitc)))
34->redo 33
(TRUE ((unit . unita)))
35->redo 33
(TRUE ((unit . unitc)))
36->..
a unit
..
unita
37->..
a unit..
unitg
38->redo 36 thru 37
unitc
unitc
39->redo 38
unita
unitg
40->redo 38 thru 39
unitc
unita
unitc
unitg
41->?? 15 thru 20


15.  ->.. UPDATE %[ update1 %]
     Failed...
16.  ->why

17.  ->.. PURGE %[ unita %]
     Failed...
```

```
18.  ->why

19.  ->.. UPDATE %[ update2 %]
     success
20.  ->.. unita : unit-contents
     failed...
```

6.2 IOC TRACE -- CAMELOT EXAMPLE

Below is a trace of a user interacting with the IOC to explore behavior of the CAMELOT specification. This is a specification of a process involving people and familial relationships. It was created for the purpose of testing and demonstrating IOC capabilities.

```
88←Read CAMELOT.GST

93←Pretty
begin
  type person(any child:person::0 through 2,sex,optional spouse:person::
                optional)
       bound print
       by=:sex,=::child;
  type sex;
  always prohibited OEDIPUS←COMPLEX there exists person |
       person:spouse=person:child;
  always prohibited INCEST there exists person.1,person.2 |
       (person.1::child=person.2::child)and(person.1:spouse=person.2);
  always prohibited SPOUSES←OF←SAME←SEX there exists person.1,
                                                person.2 |
       (person.1:spouse=person.2)and(person.1:sex=person.2:sex);

  action
    MARRY[person.1,person.2]
    definition atomic insert person.1:spouse=person.2;
                      insert person.2:spouse=person.1
             end atomic;

  action
    MAKECHILD[person]
      precondition there exists person.1 | person:spouse=person.1
    let gender:sex=a sex
    definition atomic create person.3;
                      insert person.3:sex=gender;
                      insert person:child=person.3;
                      insert person:spouse:child=person.3;
                      case gender of
                           male=>GISTPRINT["It's a boy!"];
                           female=>GISTPRINT["It's a girl!"];
                      end case
             end atomic;

  action
    DIVORCE[person.1,person.2]
      precondition person.1:spouse=person.2
    definition atomic delete person.1:spouse=person.2;
                      delete person.2:spouse=person.1
             end atomic;
```

```
  atomic insert male isa sex;
         insert female isa sex;
         insert arthur isa person;
         insert arthur:sex=male;
         insert guenevere isa person;
         insert guenevere:sex=female;
         insert lancelot isa person;
         insert lancelot:sex=male
  end atomic
end..

94←gist

              **** G I S T ****

type "help" for command list
(re)processing declarations
2->GISTPRINTDEPTH[1]
1
3->GTRTACE[T]
=GTRACE
T
4->init
inserting male isa sex in cx:2
inserting female isa sex in cx:2
inserting arthur isa person in cx:2
inserting arthur : sex = male in cx:2
inserting guenevere isa person in cx:2
inserting guenevere : sex = female in cx:2
inserting lancelot isa person in cx:2
inserting lancelot : sex = male in cx:2

5->..
MARRY[arthur,guenevere]..

MARRY:
person.1 = arthur    :sex =    male

person.2 = guenevere :sex =    female

inserting arthur : spouse = guenevere in cx:4
inserting guenevere : spouse = arthur in cx:4
MARRY completed
success
6->..
MAKECHILD[arthur]..

MAKECHILD:
person = arthur    :sex =    male

inserting per0029 isa person in cx:7
inserting per0029 : sex = male in cx:7
inserting arthur : child = per0029 in cx:7
```

```
inserting guenevere : child = per0029 in cx:7
It's a boy!
MAKECHILD completed
success
7->..
arthur:child:sex ..
male
8->..
guenevere:child:sex ..
male
9->..
male::sex..
per0029
10->redo
lancelot
11->..
per0029::sex ..
Failed...
12->..
per0029::child ..
guenevere
13->redo
arthur
14->redo 12 thru 13
guenevere
arthur
15->..
male::sex::child..
guenevere
16->redo
guenevere
17->OK
T
95←In

96←Next -1

97←Pretty
atomic insert male isa sex;
       insert female isa sex;
       insert arthur isa person;
       insert arthur:sex=male;
       insert guenevere isa person;
       insert guenevere:sex=female;
       insert lancelot isa person;
       insert lancelot:sex=male
end atomic..
98←Before
relation PARENT←IN←LAW(z:person,y:person) definition z:spouse::child=y ..

99←Top

2←gist
```

```
              **** G I S T ****

type "help" for command list
(re)processing declarations

18->GUNTRACE[T]
NIL
19->arthur:spouse

u.b.a.
arthur:spouse
20->..
arthur:spouse ..
Failed...
21->UNDO 16
redo undone.
22->..
arthur:spouse ..
guenevere
23->..
exists p1:person,p2:person|PARENT←IN←LAW(p1,p2)..
(FALSE)
24->..
MARRY[lancelot,arthur:child]..

MARRY:
person.1 = lancelot  :sex =    male

person.2 = per0029   :sex =    male
                     ::child = arthur
                               guenevere

inserting lancelot : spouse = per0029 in cx:10
inserting per0029 : spouse = lancelot in cx:10
Failed...
25->why

Constraint Violation:
always prohibited SPOUSES←OF←SAME←SEX there exists person.1,
                                                 person.2 |
       (person.1:spouse=person.2)and(person.1:sex=person.2:sex)..

Variable bindings:
((person.1 . lancelot) (person.2 . per0029))
NIL
26->..
MAKECHILD[arthur]..

MAKECHILD:
person = arthur    :sex =    male

inserting per0031 isa person in cx:12
inserting per0031 : sex = male in cx:12
inserting arthur : child = per0031 in cx:12
inserting guenevere : child = per0031 in cx:12
```

```
It's a boy!
MAKECHILD completed
success
27->redo

MAKECHILD:
person = arthur    :sex =    male

inserting per0032 isa person in cx:15
inserting per0032 : sex = female in cx:15
inserting arthur : child = per0032 in cx:15
inserting guenevere : child = per0032 in cx:15
It's a girl!
MAKECHILD completed
success
28->..
exists p1:person,p2:person|PARENT←IN←LAW(p1,p2)..
(FALSE)
29->..
MARRY[lancelot,arthur:child]..

MARRY:
person.1 = lancelot  :sex =    male

person.2 = per0032   :sex =    female
                     ::child = arthur
                               guenevere

inserting lancelot : spouse = per0032 in cx:18
inserting per0032 : spouse = lancelot in cx:18
MARRY completed
success
30->..
exists p1:person,p2:person|PARENT←IN←LAW(p1,p2)..
(TRUE ((p1 . lancelot) (p2 . guenevere)))
31->redo
(TRUE ((p1 . lancelot) (p2 . guenevere)))
32->..
loop (a person|PARENT←IN←LAW(lancelot,person)) do GISTPRINT[person]..
guenevere :sex =    female

arthur    :sex =    male

success
```

7. DESIGN FOR A GIST SPECIFICATION VALIDATION FACILITY

7.1 OBJECTIVE

The objective of a Gist Specification Validation Facility will be to determine the feasibility of validating a functional specification, treated as an operational system prototype, through observation of its behavior on both concrete and symbolic data. Existing technology will be extended to develop and demonstrate a prototype tool which aids a system design specialist in generating, documenting, and validating operational (executable) software requirements specifications.

7.2 BACKGROUND

Formal specifications act as a contract between clients and implementors. Since the implementation costs are so high and the implementation delays so long, it is crucial that the formal specification accurately portray the client's needs. Much time, money, and effort has been wasted because of mismatches between specifications and client needs.

This effort attempts to mitigate this problem by developing a methodology for using formal specifications as system prototypes and validating them through observation of their behavior produced by symbolic evaluation.

There are two key technical aspects to this approach. First, the creation of a high-level formal operational specification language and an interpreter of that language. This enables the formal specifications to be used as a prototype. Second, a symbolic evaluator of this formal specification language which enables entire classes of test cases to be explored simultaneously. Such collapsing of the space of test cases for the prototype, enables it to be "exhaustively" tested to ascertain whether it meets client needs.

Both of these capabilities exist in rudimentary form in the SAFE project at ISI, and their extraction from SAFE and extension are the foundation of this effort. Under this contract we have provided the first of these two required capabilities: the creation of a formal operational specification language and an interpreter of that language (see Chapters 2 and 5). An initial operating capability (IOC) exists (see Chapter 5) which allows a user/analyst to enter input data required by a formal specification (prototype) and "run" the prototype, monitoring selected events or breaking upon specified situations, as with normal DDT-like debuggers, except that all interactions are at the conceptual specification level, rather than the low-level representation-dependent implementation level.

Since symbolic evaluation is so central to this effort, we will explain it in more detail here before describing our approach. The purpose of symbolic execution is to expose the behavior of a specification on a class of inputs rather than just on a single test case as would occur with a normal (nonsymbolic) evaluator. This capability arises from the ability to recognize assumptions implicit in the specification and to incorporate them as fuller descriptions of the incomplete (symbolic) inputs, and from the ability to explore all conditional paths in the specification and to annotate the behavior of such paths with their conditionality.

The size of the class of test cases being explored by a symbolic execution depends upon the degree of "symbolicness" of the input and initial data base description, and upon the amount of refinement provided by the user/analyst during the symbolic execution. "Symbolic" and "Concrete" should be recognized as the endpoints of a spectrum of degrees of completeness of description. At the concrete endpoint, the description of the input (and initial data base) is complete so that all information required during evaluation is already present. Therefore evaluation can proceed in a normal (nonsymbolic) manner, with data accessed as needed and conditional branching determined by the available concrete data. A single test case execution results.

As the description of the input (and initial data base) becomes more symbolic (incomplete), some of the information needed during evaluation is not available. One of two options must be selected before evaluation can proceed:

1. The needed information is provided (dynamically) by the user/analyst.

2. The needed information is treated "symbolically" so that its presence and value are unknown and the full conditionality of the specification (with respect to this needed information) is explored.

With the first option, the symbolic evaluation is being used to incrementally define a test case, or class of test cases, of interest. The symbolic evaluation, rather than the user/analyst, determines what information is relevant for the definition of different test cases.

With the second option, a class of test cases being simultaneously explored is formed. The more this option is employed, the larger the class of test cases becomes. However, the complexity of the behavior of the specification on that class of test cases also rises proportionately so that understanding that behavior becomes correspondingly more difficult.

Thus a balance must be struck between reducing the number of classes of test cases to explore and increasing the complexity of behavior of the specification on those classes. For this reason, the

symbolic evaluator will be interactive so that the user/analyst can control the definition of test cases class size and their corresponding complexity.

A short example may help clarify these issues. The example is taken from the Program Support Library, that we have formally specified under this contract (see Chapter 3). Let's assume that we want to test the specification's behavior on insertion of lines into an existing unit. We therefore create a symbolic command leaving unspecified: the sequence of lines to insert, the unit into which the insertion is to be made, and the specification of the location at which the insertion is to occur. However, since we are only interested in the insertion behavior, we concretely specify the command type as an insertion command. With this set of choices the initial data base is merely the symbolic command that we've created and it is passed as the input to the specification.

We now start the symbolic execution. The specification first determines what type of command is being processed. Since this is concretely specified as an insertion command, execution proceeds normally. Next, the specification attempts to access the unit specified in the command. Since this was unspecified, the symbolic evaluator asks the user/analyst whether he wants to make a concrete choice or not. If the user/analyst chooses to supply a concrete unit, then he either specifies an existing one (we have assumed that none were predefined in the initial data base) or creates a new one specifying whatever attributes he desires. Such dynamic specification of concrete data is simply a need driven incremental definition of the initial data base. On the other hand, if he doesn't want to supply a concrete unit, the system will create a new symbolic unit and proceed.

After several more information-gathering steps similar to the one above, the symbolic evaluator reaches the crucial step for this class of test cases--whether the insertion point exists within the sequence of lines which constitute the contents of the unit. If both the sequence and the insertion point are concrete, then the insertion point either is in the sequence or not, and this can be determined by normal evaluation. On the other hand, if either or both values are symbolic, there is no way to make this determination. In such situations, the user/analyst is asked which of three options should be explored.

1. Pursue the TRUE branch of the predicate by constraining the symbolic value(s) so that the predicate would be TRUE.

2. Pursue the FALSE branch of the predicate by constraining the symbolic value(s) so that the predicate would be FALSE.

3. Pursue both branches by leaving the decision symbolic. This enlarges the class of test cases being explored.

Let's assume that the third option is selected by the user/analyst. The symbolic evaluator would then pursue branches separately by suitably constraining the symbolic value(s) in each branch. The activity of each branch would be marked by the conditionality of the predicate so that after the branches joined, the system could determine the conditionality of the respective behaviors and thus their interaction with later activity.

In this specific case, this dual exploration is particularly simple since one branch terminates in an error (the insertion point is not within the contents of the unit) and thus the branches do not rejoin. It's quite evident that as the size of the class of test cases being explored grows, the difficulty of clearly explaining this behavior in terms of the internal branch points increases. We expect such explanation capabilities to be the major portion of this effort.

7.3 APPROACH

This two-year effort will build upon the foundation established by the existing contract via a continued phased introduction of capabilities from the SAFE project to extend the IOC into the symbolic evaluation-based specification validation facility required.

First, SAFE capabilities to record and interrogate execution behavior will be integrated into the IOC. This is necessary because the existing IOC only maintains a dynamically changing "current state" of the world being modeled in the relational data base. Since we are interested in explaining the behavior of the system (i.e., its sequence of actions) which produced some final state, rather than merely describing the final state, recording and saving the intermediate activity is crucial. Such recording produces a data base of behavior which can be accessed to provide summarized descriptions of activity used to validate the specification. To obtain such summarized descriptions, a tool which interrogates this data base and organizes the information it contains into activity summaries (explanations of behavior) must be built. This is the major new research proposed under this contract. To gain experience with creating such summaries and to provide a fallback position in case such automatic activity summarization proves infeasible, a set of interactive user/analyst commands for examining the recorded behavior will first be provided.

At this point, a capability would exist for validating formal operational specifications by analyzing their behavior on concrete test cases. But even at the specification level, the number of concrete test cases that would have to be explored to adequately validate the specification is overwhelming. This difficulty can be overcome by integrating the symbolic execution capability from the SAFE project into this validation facility (as an extension to the language interpreter) and extending the summarization capability to include the symbolic data in the recorded behavior.

7.4 PLAN

The history recording and interrogation facility from the SAFE system will be extracted and integrated into the IOC. This facility records execution behavior and provides low-level operations for examining this behavior. Then, a set of high-level user (analyst) commands for examining the recorded behavior will be provided. These commands should provide the ability for the user to move back and forth in the history by describing desired states (via GIST specification language predicates). These commands should also provide the analyst with the ability to determine which portions of the specification are responsible for each piece of behavior.

Next a tool which analyzes a recorded history and provides a summary description will be built. This tool will use the structure of the specification and rules of program description to summarize the recorded behavior.

Then, the symbolic execution package from SAFE will be extracted and modified so that it interactively receives guidance from the user/analyst as to which execution paths to explore. A bookkeeping facility to keep the user/analyst informed as to which paths have been explored and which have not yet been explored will also be provided.

Finally, the automatic summarization tool will be extended to include descriptions of symbolic behavior.

Appendix I
GIST GRAMMAR

Below is the grammar of Gist used in generating the IOC. The form of the grammar is that accepted by the POPART system. The grammar is context free, represented as a series of productions of the form

nonterminal : = rewrite

with the following conventions:

- The rewrite for a nonterminal symbol is a sequence of nonterminal symbols and terminal symbols.
- Terminal symbols are preceded by a quote mark (').
- Alternative rewritings of a nonterminal are separated by bars (|).
- Lines of the form

  NOTE ... ;

  are comments about the grammar, not productions of the grammar.
- Optional constituents are enclosed in braces {}.
- The nonterminal LEXEME is predefined to allow any sequence of letters and digits as its rewrite.
- A rewrite of the form

  nonterminal |> predicate

  indicates that subset of rewrites of the nonterminal which satisfy the named unary predicate. The predicates are not documented here.

```
NOTE 1 Declaration section;

DeclarationStatement := IndividualDeclaration
                        { LocalDefinitions }
                        { IndividualDefinition }
                        { ConstrainingDeclarations };
IndividualDeclaration := TypeDeclaration |
                         BoundaryDeclaration |
                         RelationDeclaration |
                         ProhibitDeclaration |
                         RequireDeclaration |
                         DerivationDeclaration |
                         ActionDeclaration |
                         DemonDeclaration  ;

LocalDefinitions := 'let LocalDefinition + ', ;
```

```
  LocalDefinition := Role '= ObjectExpression;
IndividualDefinition := 'definition (Statement | Predicate | TypeExpression |
                                    ObjectExpression );

ConstrainingDeclarations := 'where DeclarationOrComment + '; { '; } 'end;
  DeclarationOrComment := DeclarationStatement | CommentStatement  ;


BoundaryDeclaration := 'boundary { Name } BoundarySpecification;
  BoundarySpecification := Linkage + ', ;
    Linkage := LinkClass Link;
      LinkClass := EquivalenceName | Predicate  ;
        EquivalenceName := '= | 'equiv | 'identical ;
      Link := AttributeRelationship AttributeName;
        AttributeRelationship := ': | ':: ;

TypeDeclaration := 'type { Name } { '( AttributeSpec + ', ') }
                          { TypeRelationQualifier + };
  AttributeSpec := { CountSpec } AttributeName
                   { ': TypeExpression } { ':: CountSpec# };
    AttributeName := RoleName  ;
    CountSpec := CountSpecConstant | IntegerRange | Integer |
                     NestedCountSpec ;
        IntegerRange := Integer 'through Integer#;
        NestedCountSpec := '( CountSpec + ', ') ;
      CountSpecConstant := 'unique | 'optional | 'multiple | 'any;
      TypeRelationQualifier := TypeQualifier | RelationQualifier ;
      Name := LEXEME |> IdentifierFilter ;
      Identifier := LEXEME |> IdentifierFilter ;
        TypeExpression :=  TypePrimitive  ;
          TypePrimitive :=  StructureSpecification |
                              SetTerm  |
                            DeclarationStatement  ;
          StructureSpecification := StructureType  'of TypeExpression;
            StructureType := PredefinedStructure | Identifier ;
                PredefinedStructure := 'set | 'sequence ;

RelationDeclaration := 'relation { Name } '( Role + ', ')
                       { RelationQualifier + } ;
    Role := RoleName { ': TypeExpression } ;
      RoleName := Identifier { '. Distinguisher }  ;
        Distinguisher := IntegerConstant | Identifier  ;
          IntegerConstant := LEXEME |> Integer? ;
          Integer := LEXEME |> Integer? ;


ProhibitDeclaration := 'always 'prohibited { Name } Predicate;

RequireDeclaration := 'always 'required { Name } Predicate;

  DerivationDeclaration := 'derivation { Name } '( Role + ', ')
              'whenever Predicate 'conclude Pattern + ', ;

  ActionDeclaration := 'action { Name } { '[ { Role + ', } '] }
                       { ActionQualifier + } ;
```

```
  ActionQualifier := ActionPrecondition |
                     ActionPostcondition  ;
    ActionPrecondition := 'precondition Predicate;
    ActionPostcondition := 'postcondition Predicate;

  DemonDeclaration := 'demon { Name } '( Role + ', ')
                      'trigger Predicate
                      'response Statement ;

NOTE 2 Statement definitions;

Statement := DeclarationStatement |
             RequireStatement |
             ProhibitStatement |
             DataPrimitiveStatement |
             AtomicStatement |
             BlockStatement |
             ConditionalStatement |
             CaseStatement |
             ActionInvocation |
             LoopingStatement |
             CommentStatement |
             ContinuationStatement |
             AttemptStatement |
             ChooseStatement |
             TogetherStatement ;

  RequireStatement := 'require Predicate;

  ProhibitStatement := 'prohibit Predicate;
  DataPrimitiveStatement := DataInsertion | DataDeletion | DataUpdate |
                            ObjectCreation | ObjectDestruction | ObjectCopy  :

    DataInsertion := 'insert { All } Pattern;
    DataDeletion := 'delete { All } Pattern;
    DataUpdate := 'update { All } { AttributeRelationship }
                          RoleName OfExpressionOrRelation 'to Expression;
      OfExpressionOrRelation := 'in Relationship | 'of ObjectExpression ;
    ObjectCreation := 'create CreationSpecification + ',
                      { 'with Pattern } ;
      CreationSpecification := RoleName { ': NameOrPattern } { 'wrt Boundary};
        NameOrPattern := Pattern | RoleName  ;
          Pattern := Identity | Containment | Relationship  ;
    ObjectDestruction := 'destroy { All } ObjectExpression { 'wrt Boundary };
    ObjectCopy := 'copy CreationSpecification + ', ;
      All := 'all ;

  AtomicStatement := 'atomic Statement + '; 'end 'atomic;
  BlockStatement := 'begin Statement + '; 'end;

  ConditionalStatement := 'if Predicate 'then Statement
                          { ElseIfClause + } { 'else Statement# };
  CaseStatement := 'case ObjectExpression 'of CaseCase + '; { '; }
                   'end 'case ;
    CaseCase := ObjectExpression  '=> Statement;
```

```
    ElseIfClause := 'elseif Predicate 'then Statement;
  LoopingStatement := 'loop SetOrSequenceExpression 'do Statement;
  CommentStatement := 'comment AnythingButEnd + 'end 'comment ;
    AnythingButEnd := LEXEME |> AllButEnd;
  ContinuationStatement := 'continuations 'of Statement
                           'from 'which Expression
                           'satisfies Predicate;
  AttemptStatement := 'attempt Statement + '; ;
  ActionInvocation := Name '[ { Expression + ', } '];
  ChooseStatement := 'choose Statement + '; 'end 'choose;
  TogetherStatement := 'together Statement + '; 'end 'together;

NOTE 3 The extensions section;

TypeQualifier := BoundaryDefault | TypeEmbedding |
                 EquivalentForm  ;
  BoundaryDefault := 'bound BoundaryOp + ', 'by Boundary;
    BoundaryOp := 'copy | 'destroy | 'restore | 'contains | 'equiv |
                  'modify | 'all;
    Boundary := BoundarySpecification | Name  ;

  TypeEmbedding := TypeConnector 'of '< TypeDeclarationBody + '; '> ;
    TypeConnector := { CountSpec } Supertype | Subtype;
      Subtype := 'subtype;
      Supertype := 'supertype;
  EquivalentForm := LinkClass;

RelationQualifier := KeySpecification | CoversSpecification |
                     RelationMappingProperty  ;
  KeySpecification := 'key KeyRoles;
    KeyRoles := Role | '( ObjectExpression + ', ') ;
  CoversSpecification := 'covers Roles;
    Roles := Role | '( Role + ', ') ;
    TypeDeclarationBody := Name { '( AttributeSpec + ', ') }
                               { TypeRelationQualifier + }
                               { LocalDefinitions }
                               { IndividualDefinition }
                               { ConstrainingDeclarations } ;
  RelationMappingProperty := ' onetoone | ' onto | 'reflexive |
                         'transitive | 'symmetric | 'antisymmetric |
                         'nonreflexive ;
NOTE 4  Expression language;
Expression := ObjectExpression  ;
SetOrSequenceExpression := SetExpression | SequenceTerm  ;
ObjectExpression := SelectedExpression | SetExpression |
                        SequenceExpression | ArithmeticExpression |
                       SharedExpression   ;


SharedExpression := SharedFactor  ;
SharedFactor := SharedSecondary { 'asof Duration }  ;
SharedSecondary := { OldConstant } SharedPrimary  ;
  OldConstant := 'old ;
SharedPrimary := SharedPrimitive
                             { Selectors }  ;
```

```
  Selectors := Selector + ;
  Selector := AttributeRelationship AttributeName;
SharedPrimitive := FunctionCall | SpecialSymbol |
                          NestedObjectExpression | Variable  ;

  FunctionCall := Identifier '( { ObjectExpression + ', } ') ;
  SpecialSymbol := ValueSelectedSymbol | WildCardSymbol | ;
    ValueSelectedSymbol := '* ;
    WildCardSymbol := '$ ;

  NestedObjectExpression := '( ObjectExpression ') ;
  Variable := RoleName  ;


SelectedExpression := SequenceSelection | GeneratorSelection |
                      NondeterministicSelection | OptimalSelection  ;
  SequenceSelection := RandomSelection | FirstElement | LastElement  ;
    RandomSelection := SequenceTerm 'sub ArithmeticTerm ;
    FirstElement := 'first SequenceTerm;
    LastElement := 'last SequenceTerm;
  GeneratorSelection := GeneratorTerm LastWord ObjectExpression;
    LastWord := 'afterwards | 'lasttime ;
  NondeterministicSelection := Determiner Role { '| Predicate } ;
    Determiner := 'a | 'an | 'any | 'one | 'the  ;
  OptimalSelection := OptimizingWord ObjectExpression 'wrt Order;
    OptimizingWord := 'maximal | 'minimal ;

NOTE  Set expression subgrammar;

SetExpression := SetTerm  |> RejectSharedExpression ;
SetTerm := SetFactor { DifferenceSetTerm }  ;
  DifferenceSetTerm := '- SetTerm ;
SetFactor := SetSecondary { UnionSetFactor }  ;
  UnionSetFactor := 'union SetFactor ;
SetSecondary := SetPrimary { IntersectSetSecondary }  ;
  IntersectSetSecondary := 'intersect SetSecondary ;
SetPrimary := { Powerset } SetPrimitive  ;
  Powerset := 'powerset ;
SetPrimitive := EnumeratedSet | SetRestriction | GeneratorTerm  ;
  EnumeratedSet := '{ ObjectExpression + ', '} ;
  SetRestriction := '{ Role '| Predicate '} ;

NOTE  Sequence Expression subgrammar;

SequenceExpression := SequenceTerm  |> RejectSharedExpression ;
SequenceTerm := SequenceFactor { ConcatenateSequenceTerm }  ;
  ConcatenateSequenceTerm := ' SequenceTerm ;
SequenceFactor := SequenceSecondary { LimiterSequenceFactor }  ;
  LimiterSequenceFactor := Limiter SequenceFactor;
  Limiter := 'before | 'beyond ;
SequenceSecondary := EnumeratedSequence | String |
                     GeneratorTerm  ;
  EnumeratedSequence := '< ObjectExpression + ', '> ;
  String := LEXEME |> CollectString;
```

```
NOTE Generator expressions subgrammar;

GeneratorExpression := GeneratorTerm  |> RejectSharedExpression ;
GeneratorTerm := GeneratorFactor { TogetherGeneratorTerm }  ;
  TogetherGeneratorTerm := Together GeneratorTerm ;
  Together := 'with | 'cross ;
GeneratorFactor := GeneratorSecondary { GeneratorSuffixes }  ;
  GeneratorSuffixes := GeneratorSuffix + ;
    GeneratorSuffix := PredicatedFilter | EncodingFilter |
                       NamingFilter | SequenceTerminator |
                       Accumulation | Remainder | SubsequenceSelection   ;
    PredicatedFilter := SuchthatOrUnless Predicate;
      SuchthatOrUnless := 'suchthat | 'unless ;
    EncodingFilter := 'obtain ObjectExpression ↑ ', ;
    NamingFilter := 'named Variable { CommaVariable + 'respectively } ;
      CommaVariable := ', Variable  ;
    SequenceTerminator := TerminatorWord Predicate;
      TerminatorWord := 'while | 'until ;
    Accumulation := { 'initially ObjectExpression }
                    'accumulate ObjectExpression# ;
    Remainder := 'ungenerated | 'rest ;
    SubsequenceSelection := DurationWord Predicate;
      DurationWord := 'before | 'through | 'after | 'beyond ;
GeneratorSecondary := PrimitiveGenerator | PrimitivePatternGenerator |
                      SharedExpression  ;
  PrimitiveGenerator := 'from ObjectExpression 'by ObjectExpression#;
  PrimitivePatternGenerator := SequenceTerminator  ;

NOTE Arithmetic expression subgrammar;

ArithmeticExpression := ArithmeticTerm  |> RejectSharedExpression ;
ArithmeticTerm := ArithmeticFactor { PlusArithmeticTerm }  ;
  PlusArithmeticTerm := PlusOp ArithmeticTerm;
  PlusOp := '+ | '- ;
ArithmeticFactor := ArithmeticSecondary { MultiplyArithmeticFactor }  ;
  MultiplyArithmeticFactor := MultOp ArithmeticFactor ;
  MultOp := 'x | '/ | 'div | 'mod ;
ArithmeticSecondary := { ArithmeticPrefixes } ArithmeticPrimary  ;
  ArithmeticPrefixes := ArithmeticPrefix + ;
  ArithmeticPrefix := '+ | '- | 'ceiling | 'floor | 'absolute ;
ArithmeticPrimary := ArithmeticPrimitive { ExponentArithmeticSecondarys }  ;
  ExponentArithmeticSecondarys := ExponentArithmeticSecondary + ;
  ExponentArithmeticSecondary := '** ArithmeticSecondary ;
ArithmeticPrimitive := Integer | SharedExpression  ;

NOTE  Predicate expression subgrammar;

Predicate := QuantifiedLogical | DurationPredicate | LogicalImp  ;
QuantifiedLogical := { Not } QuantHead Predicate;
QuantHead := ExHead | UnivHead ;
  ExHead := { 'there } 'exists QuantifierRole ↑ ', '| ;
  UnivHead := { 'for } 'all QuantifierRole ↑ ', ':: ;
    QuantifierRole := Role  ;
DurationPredicate := { Not } AlwaysOrNever Predicate 'throughout Duration;
```

```
  AlwaysOrNever := 'always | 'never ;
LogicalImp := LogicalTerm { 'implies LogicalImp }  ;
LogicalTerm := LogicalFactor { OrLogicalTerm }  ;
  OrLogicalTerm := Or LogicalTerm ;
LogicalFactor := LogicalSecondary { 'and LogicalFactor }  ;
  Or := 'or | 'xor ;
LogicalSecondary := { LogicalPrefixes } LogicalPrimary  ;
  LogicalPrefixes := LogicalPrefix + ;
    LogicalPrefix := Not | OldConstant  ;
  Not := 'not | '~ ;
LogicalPrimary := Proposition { 'asof Duration }  ;

Proposition := LogicalConstant | InfixRelation |
        Relationship |   NestedPredicate  ;
  LogicalConstant := 'TRUE | 'FALSE | 'true | 'false ;
  InfixRelation := ObjectExpression ComparisonSuffix ;
    ComparisonSuffix := OrderingRelation | OptimizationRelation |
                        IdentityRelation | ContainmentRelation  ;
  OrderingRelation :=  { Immediately }
                         OrderingRelationConstant ObjectExpression
                           { 'wrt Order } ;
    Immediately := 'immediately ;
    OrderingRelationConstant := '< | '> | 'tie ;
    Order := Identifier  ;
  OptimizationRelation :=  Optimizes ObjectExpression
                          { 'wrt Order } ;
    Optimizes := 'minimizes | 'maximizes ;
  Identity :=  '= ObjectExpression ;
  Relationship := FunctionCall ;
  Containment :=  ContainmentName
               ( ObjectExpression | TypeExpression ) ;
    ContainmentName := ElementName | ProperSubSuperSet |
                     ProperSubSuperSequence  ;
    ElementName := 'contains | 'in | 'isa | 'isan ;
  ProperSubSuperSet := { Proper } SubOrSuperSet  ;
  ProperSubSuperSequence := { Proper } SubOrSuperSequence  ;
    SubOrSuperSet := 'subset | 'superset ;
    Proper := 'proper ;
    SubOrSuperSequence := 'subsequence | 'supersequence ;
NestedPredicate := '( Predicate ') ;


Duration := GeneratorTerm  ;
```

Appendix II
GIST SPECIFICATIONS

Within this appendix:

comments to explain language features are surrounded by !!!

comments to explain modeling of the task are surrounded by ***

II.1 SOURCE DATA MAINTENANCE SPECIFICATION

II.1.1 *** Type definitions for objects of domain ***

```
type project(LIBRARY :any ::unique , CLASSIFICATION)
                bound contains by :LIBRARY
                bound restore by = :CLASSIFICATION,
                                 ≡ :LIBRARY ;

type library(FILE :any ::unique)
                bound contains, restore by ≡ :FILE ;

type file(UNIT :any ::unique , PASSWORD :optional)
                bound contains by :UNIT
                bound restore by ≡ UNIT,
                                 = :PASSWORD ;

type unit(UNIT-CONTENTS | sequence of line , PROGRAMMER ,
      UNIT-KEY | key :optional , SOURCE-LANGUAGE | language :optional)
                bound contains by :UNIT-CONTENTS
                bound ≡, restore by ≡ :UNIT-CONTENTS,
                                 = :PROGRAMMER,
                                 = :UNIT-KEY,
                                 = :SOURCE-LANGUAGE ;

type line(CHARS | sequence of character , LINE-INCLUDES-UNIT | essence of unit :any)
                bound restore by ≡ :CHARS,
                                 = :LINE-INCLUDES-UNIT ;

type password(); type programmer(); type language();
type key(); type classification();
```

II.1.2 *** Static constraints on the world ***

*** These constrain the possible states of the world (without reference to changes in the world, hence 'static').
***

always required UNIT-LINE-LENGTH-CONSTRAINT
*** All lines in units are of length Card-Length ***
∀ *line* || MEMBER($:UNIT-CONTENTS,$,*line*) ⇒ (length(*line*:CHARS) = *Card-Length*) ;

always prohibited
*** A line may not occur more than once in the unit-contents sequence ***
∃ *unit* , *integer*.1 , *integer*.2 ||
MEMBER(*unit*:UNIT-CONTENTS,*integer*.1,*) = MEMBER(*unit*:UNIT-CONTENTS,*integer*.2,*) ;

always prohibited
*** A line may not be shared between units ***
∃ *unit*.1, *unit*.2 ||
MEMBER(*unit*.1:UNIT-CONTENTS , $, *) = MEMBER(*unit*.2:UNIT-CONTENTS , $, *) :

*** Inclusion of units may not be 'circular' - define a relation and derivations to model inclusion between units ***

relation UNIT-INCLUDES-UNIT(*includor* | unit , *included* | unit) ;

derivation DERIVE-UNIT-INCLUDES-UNIT(*unit-a* | unit , *unit-b* | unit)
*** If a line in unit-a includes unit-b, derive that unit-a UNIT-INCLUDES-UNIT unit-b ***
whenever ∃ *line* || MEMBER(*unit-a*:UNIT-CONTENTS,$,*line*) ∧ (*line*:LINE-INCLUDES-UNIT = *unit-b*)
conclude UNIT-INCLUDES-UNIT(*unit-a*,*unit-b*) ;

derivation DERIVE-TRANS-CLOSURE-UNIT-INCLUDES-UNIT(*unit-a* | unit , *unit-c* | unit)
*** Form transitive closure of UNIT-INCLUDES-UNIT ***
whenever ∃ *unit-b* | unit ||
UNIT-INCLUDES-UNIT(*unit-a*,*unit-b*) ∧ UNIT-INCLUDES-UNIT(*unit-b*,*unit-c*)
conclude UNIT-INCLUDES-UNIT(*unit-a*,*unit-c*) ;

*** Now prohibit any circular UNIT-INCLUDES-UNIT ***
always prohibited ∃ *unit* || UNIT-INCLUDES-UNIT(*unit*,*unit*) ;

II.1.3 *** User commands ***

*** We have chosen to model user commands as a sequence of objects through which the system iterates (an alternative would have been to expect the world to invoke the appropriate action for each user command).
***

*** *file-spec is introduced to connect a file reference with a password* ***

```
type  file-spec(FILE, PASSWORD :optional) ;
```

*** Definitions are indented to display the type hierarchy of the commands ***

```
type  command() supertype of
      < update-command ; copy-file-command ; backup-command ; restore-command > ;

type     update-command(CURRENT-FILE-SPEC | file-spec ,
               UPDATE-SUB-COMMANDS | sequence of update-sub-command) ;

type update-sub-command(CURRENT-UNIT | essence of unit, UNIT-KEY | key :optional) supertype of
          < add-command ; purge-command ; replace-command ;
               copy-unit-command ; change-command > ;


type       add-command(DATA | sequence of line, SOURCE-LANGUAGE | language :optional) ;

type         purge-command();

type         replace-command(DATA | sequence of line) ;

type      copy-unit-command(FROMUNIT | essence of unit, FROM-FILE-SPEC | file-spec :optional) ;

type         change-command(CHANGE-SUB-COMMANDS | sequence of change-sub-command) ;

type         temporary-change-command subtype of < change-command > ;

type           change-sub-command() supertype of
                  < insert-command ; delete-command ;  shift-command ;
                   modify-command , copy-lines-command ; replace-line-command > ;
```

```
type        insert-command(DATA | sequence of line, SEQUENCE-LOCATION :optional) ;

type        delete-command(LINES | essence of sequence of line) ;

type        shift-command(LINES | essence of sequence of line, COLUMNS | integer) ;

type        modify-command(LINES | essence of sequence of line,
                    NEW-STRING | sequence of character,
                    STARTING-COLUMN | integer :optional,
                    OLD-STRING | sequence of character :optional) ;

type        copy-lines-command(FROMUNIT | essence of unit,
                LINES | essence of sequence of line,
                SEQUENCE-LOCATION, FROM-FILE-SPEC | file-spec :optional) ;

type        replace-line-command(OLDLINE | essence of line, NEWLINE | line) ;


type    copy-file-command(TO-FILE-SPEC | file-spec, FROM-FILE-SPEC | file-spec) ;


type    backup-command(OBJECT | essence of  project ∪ library ∪ file) ;

type    restore-command(BACKUP-STATE | state ,
                OBJECT | essence of  project ∪ library ∪ file ∪ unit) ;
```

*** Note that some of the names of the above commands, and the actions that are to perform them, do not correspond to the user command names given in the English specification. These changes are intended to clarify the action of the named commands.
COPY has become COPY-LINES
MOVE has become COPY-UNIT
MERGE has become COPY-FILE
Whereas previously providing sequenced numbered data lines in a CHANGE command implicitly caused insertion or replacement of lines, now these changes are achieved by giving explicit commands INSERT and REPLACE-LINE.
***

II.1.4 *** Dynamic constraints on the world ***

always prohibited **INCLUDED-UNIT-DELETION**
*** A unit may not be deleted if it is included anywhere ***
∃ *dead-unit* | essence of **unit** ||
destroyed *dead-unit* ∧ **UNIT-INCLUDES-UNIT**($,*dead-unit*) ;

always required **UNIT-KEY-CHECKING**
*** May not make changes to a keyed unit without having provided the correct key in the current command, except when performing RESTORE or RESTORE-UNITS-TEMPORARILY-CHANGED-SINCE. ***
(∃ *changed-unit* | essence of **unit** ||
(destroyed(*unit*) ∨ modified(*unit*)) ∧ old { ∃ *key* = *unit*:UNIT-KEY }) ∧
~ (RESTORE[$] ∨ RESTORE-UNITS-TEMPORARILY-CHANGED-SINCE[$]) ⇒
∃ *command, event* ||
active *event* ∧ (parameterof(*event*) = *command*) ∧ *command*:UNIT-KEY = *key* ;

always required **PASSWORD-CHECKING**
*** May not access any part of a password-protected file without having provided the correct password in the file-spec, except when performing RESTORE or RESTORE-UNITS-TEMPORARILY-CHANGED-SINCE. ***
(∃ *file* || access(*file*) ∧ ∃ *password* || *password* = *file*:PASSWORD) ∧
~ (RESTORE[$] ∨ RESTORE-UNITS-TEMPORARILY-CHANGED-SINCE[$]) ⇒

∃ *command, event, file-spec* ||
active *event* ∧ (parameterof(*event*) = *command*) ∧ *command* contains *file-spec* ∧
(*file-spec*:FILE = *file*) ∧ (*file-spec*:PASSWORD = *password*) ;

II.1.5 *** Interface with user ***

```
demon USER-INTERFACE(commands)
 trigger *** Place here user activity to provide commands and initiate processing ***
 response SOURCE-DATA-MAINTENANCE(commands) ;
```

II.1.6 *** Actions to perform commands ***

II.1.6.1 *** SOURCE-DATA-MAINTENANCE ***

```
action SOURCE-DATA-MAINTENANCE[commands | sequence of command]
definition
  over commands named com
  do attempt
     case com of

       update-command => UPDATE[com];

       copy-file-command => COPY-FILE[com];

       backup-command => BACKUP[com];

       restore-command => RESTORE[com];

     end case ;
```

II.1.6.2 *** UPDATE ***

```
action UPDATE[update-command]
  let file = update-command:CURRENT-FILE-SPEC:FILE
  definition
    over update-command:UPDATE-SUB-COMMANDS named up-com
    do begin
         require ∃ current-unit | essence of unit || current-unit = up-com:CURRENT-UNIT;
         attempt
           case up-com of

             add-command => ADD[current-unit,file,up-com];

             purge-command => PURGE[current-unit];

             replace-command => REPLACE[current-unit,file,up-com];

             copy-unit-command => COPY-UNIT[current-unit,file,up-com];

             change-command => CHANGE[current-unit,up-com];

           end case
       end ;
```

II.1.6.3 *** ADD ***

```
 action ADD[unit | essence of unit, file, add-command]
 definition
   begin
     atomic
       create unit, copy-of-lines | sequence of line ||
                    copy-of-lines ≡ add-command:DATA ∧
                    unit:UNIT-CONTENTS = copy-of-lines;
       insert file:UNIT = unit
     end atomic;
     if ∃ key || key = add-command:UNIT-KEY then insert unit:UNIT-KEY = key
   end ;
```

II.1.6.4 *** PURGE ***

```
action PURGE[unit]
definition destroy unit ;
```

II.1.6.5 *** REPLACE ***

```
action REPLACE[unit | essence of unit, file, replace-command]
let data | sequence of line = replace-command:DATA
definition *** If referenced unit already exists, replace its contents,
              otherwise create a new unit. ***
  if ∃ unit
  then begin
       create new-lines | sequence of line || new-lines ≡ data;
       update :UNIT-CONTENTS of unit to new-lines
     end
  else begin
       create add-command || add-command:DATA = data;
       ADD[unit,file,add-command]
     end ;
```

II.1.6.6 *** COPY-UNIT ***

```
action COPY-UNIT[unit | essence of unit, file, copy-unit-command]
definition
  atomic
    create unit || unit ≡ from-unit;
    insert file:UNIT = unit
  end atomic ;
```

II.1.6.7 *** CHANGE ***

```
action CHANGE[unit,change-command]
definition
  over change-command:CHANGE-SUB-COMMANDS named ch-com
  do attempt
    case ch-com of

      insert-command => INSERT[unit,ch-com];

      delete-command => DELETE[unit,ch-com];

      shift-command => SHIFT[ch-com];

      modify-command => MODIFY[ch-com];

      copy-lines-command => COPY-LINES[unit,ch-com];

      replace-line-command => REPLACE-LINE[unit,ch-com];

    end case ;
```

II.1.6.8 *** INSERT ***

*** When inserting lines into a unit, we may either explicitly say where the insertion is to take place. or by default let the lines be inserted after the last position of change to that unit (during the current CHANGE command), or at the start of the unit if there has been no such change.

To find the last position of change we define two relations. LATEST-STATE-OF-CHANGE(unit,state) relates a unit to the latest state (since starting execution of CHANGE) at which the unit contents then differ from the unit contents now. LATEST-POSITION-OF-CHANGE relates the unit to the position of this change.
***

```
relation LATEST-STATE-OF-CHANGE(UNIT , CHANGESTATE | state)
  definition
    changestate = latest some state || unit:UNIT-CONTENTS ~≡ ( unit:UNIT-CONTENTS as of state ) ∧
                  state after latest start CHANGE[unit,$] ;
```

```
relation LATEST-POSITION-OF-CHANGE(unit , sequence-location)
 definition
  ∃ changestate | state, longest-same-bottom | sequence of line ||
     longest-same-bottom = longest some same-bottom sequence of line ||
        LATEST-STATE-OF-CHANGE(unit,changestate) ∧
        unit:UNIT-CONTENTS = $ @ sequence-location @ same-bottom ∧
        ( unit:UNIT-CONTENTS as of changestate ) = $ @ sequence-location @ same-bottom


action INSERT[unit, insert-command]
 let data = insert-command:DATA
 definition
  if ∃ sequence-location || sequence-location = insert-command:SEQUENCE-LOCATION
  then POSITIONED-INSERT[unit,data,sequence-location]
  else UNPOSITIONED-INSERT[unit,data] ;


action POSITIONED-INSERT[unit, data | sequence of line, sequence-location]
 precondition ∃ topseq | sequence of line, bottomseq | sequence of line ||
        unit:UNIT-CONTENTS = topseq @ sequence-location @ bottomseq
 definition
  begin
   create new-lines | sequence of line || new-lines ≡ data;
   update :UNIT-CONTENTS of unit to topseq @ new-lines @ bottomseq
  end ;


action UNPOSITIONED-INSERT[unit,data | sequence of line]
 definition
  begin
   create new-lines | sequence of line: || new-lines ≡ data;
   update :UNIT-CONTENTS of unit
   to ( if ∃ topseq | sequence of line, bottomseq | sequence of line, sequence-location ||
          LATEST-POSITION-OF-CHANGE(unit,sequence-location) ∧
          unit:UNIT-CONTENTS = topseq @ sequence-location @ bottomseq
      then topseq @ new-lines @ bottomseq
      else *** if unit as yet unchanged, insert at front ***
          new-lines @ unit:UNIT-CONTENTS
      )
  end ;
```

II.1.6.9 *** DELETE ***

```
action DELETE[unit, delete-command
precondition ∃ startseq | sequence of line, endseq | sequence of line ||
        unit:UNIT-CONTENTS = startseq @ delete-command:LINES @ endseq
definition
  update :UNIT-CONTENTS of unit  to startseq @ endseq
```

II.1.6.10 *** SHIFT ***

```
action SHIFT[shift-command]
definition
  over shift-command:LINES named line
  do update :CHARS of line to
          some newchars sequence of character ||
            ISSHIFTED(newchars,chars,shift-command:columns) ;


relation ISSHIFTED(NEWCHARS | sequence of character,
          OLDCHARS | sequence of character, COLUMNS-TO-SHIFT | integer)
  definition
    ∃ old-left-blanks | sequence of " ", old-right-blanks | sequence of " ",
       new-left-blanks | sequence of " ", new-right-blanks | sequence of " ",
       non-blanks | sequence of character ||

      ~MEMBER(non-blanks , $ , " ") ∧
      oldchars = old-left-blanks @ non-blanks @ old-right-blanks ∧
      newchars = new-left-blanks @ non-blanks @ new-right-blanks ∧
      length(new-left-blanks) - length(old-left-blanks) = columns-to-shift ;
```

*** e.g. < "A" , "B" , "C" , " " , " " > may be shifted by 2 columns to give
< " " , " " , "A" , "B" , "C" > or
< " " , " " , "A" , "B" , "C" , " " > or
< " " , " " , "A" , "B" , "C" , " " , " " > etc.
***

II.1.6.11 *** MODIFY ***

```
action MODIFY[modify-command]
 let lines | sequence of line = modify-command:LINES
 let new-string | sequence of character = modify-command:NEW-STRING
 definition
   begin
     if ∃ starting-column | integer || starting-column = modify-command:STARTING-COLUMN
     then over lines named line do MODIFY-BY-COLUMN[line,starting-column,new-string];

     if ∃ old-string | sequence of character || old-string = modify-command:OLD-STRING
     then over lines named line do MODIFY-BY-REPLACEMENT[line,old-string,new-string]
   end ;


action MODIFY-BY-COLUMN[line, starting-column | integer, new-string | sequence of character]
 definition
   update :CHARS of line
     to  ( from 1 by +1 while < starting-column named i
                 obtain MEMBER(line:CHARS,i,*) ) @
         new-string @
         ( from starting-column + length(new-string) + 1 by +1 while =< length(line:CHARS)
                 named i obtain MEMBER(line:CHARS,i,*) ) ;



action MODIFY-BY-REPLACEMENT[line, old-string | sequence of character,
                             new-string | sequence of character]
 definition
   if line:CHARS = $ @ old-string @ $
   then update :CHARS of line
        to some newchars | sequence of character ||
             newchars @ $ =
               REPLACE-ALL-INSTANCES(chars,old-string,new-string) @
               some blank-seq | sequence of " ";



function
  REPLACE-ALL-INSTANCES(chars | sequence of character, old-string | sequence of character,
                        new-string | sequence of character)
   if ∃ leftchars | sequence of character, rightchars | sequence of character ||
        chars = leftchars @ old-string @ rightchars
   then REPLACE-ALL-INSTANCES(leftchars,old-string,new-string) @ new-string @
          REPLACE-ALL-INSTANCES(rightchars,old-string,new-string)
   else chars ;
```

II.1.6.12 *** COPY-LINES ***

```
action COPY-LINES[unit,copy-lines-command]
let lines | sequence of line = copy-lines-command:LINES
precondition copy-lines-command:FROMUNIT:UNIT-CONTENTS = $ @ lines @ $
definition
  begin
    require ∃ startseq | sequence of line, endseq | sequence of line ||
        unit:UNIT-CONTENTS = startseq @ sequence-location @ endseq;
    create new-lines | sequence of line || new-lines ≡ lines;
    update :UNIT-CONTENTS of unit to startseq @ ( new-lines ) @ endseq
  end ;
```

II.1.6.13 *** REPLACE-LINE ***

```
action REPLACE-LINE[unit, replace-line-command]
precondition MEMBER(unit:UNIT-CONTENTS,$,replace-line-command:OLDLINE)
definition
  update :UNIT-CONTENTS of unit
   to (unit:UNIT-CONTENTS named line
        obtain ( if line = oldline then newline else line ) ) ;
```

II.1.6.14 *** COPY-FILE ***

```
action COPY-FILE[copy-file-command]
let file = copy-file-command:TO-FILE-SPEC:FILE
let from-file | file = copy-file-command:FROM-FILE-SPEC:FILE
definition
  over from-file:UNIT named from-unit
  do atomic
       create new-unit | unit || new-unit ≡ from-unit;
       insert file:UNIT = new-unit
     end atomic ;
```

II.1.6.15 *** BACKUP and RESTORE ***

action BACKUP[*backup-command*] ;

*** Simply activating BACKUP with its parameter suffices to serve as an event that we may later refer to in a RESTORE command. ***

action RESTORE[*restore-command*]
let *obj* | essence of **project** ∪ **library** ∪ **file** ∪ **unit** = *restore-command*:OBJECT

precondition *** A backup must have been performed at the time specified in the restore command and the object being restored must have been the object (or part of the object) backed up at that time. ***
∃ *backup-command* || *restore-command*:BACKUP-STATE = start BACKUP[*backup-command*] ∧
(*backup-command*:OBJECT contains *obj*) as of *restore-command*:BACKUP-STATE

definition
restore *obj* as of *restore-command*:BACKUP-STATE;

*** Note that because RESTORE commands refer only to times at which BACKUP commands were executed, the implementor need only allow for restoration to such points in time rather than to arbitrary points during the past execution. ***

II.1.7 *** Restoration following temporary changes ***

demon TRIGGER-TEMPORARY-RESTORATION(*event*)
*** At end of job, trigger restoration of units temporarily changed during job ***

trigger ~ SOURCE-DATA-MAINTENANCE[$]
response RESTORE-UNITS-TEMPORARILY-CHANGED-SINCE[
latest start SOURCE-DATA-MAINTENANCE[$] ;

---

relation TEMPORARY-CHANGE-BEGAN(*unit,state*)
definition
∃ *tc-com* | temporary-change-command || start CHANGE[*unit,tc-com*] = *state*

action RESTORE-UNITS-TEMPORARILY-CHANGED-SINCE[*s* state]
*** Action to do restoration of all temporarily changed units to the state they were in prior to their first temporary change since state s ***
definition
over state named *s1* when *s1* = earliest some *state* || *state* after *s* ∧
(∃ *unit* || TEMPORARY-CHANGE-BEGAN(*unit,state*))
do restore *unit* as of *s1* ;

always required PRESERVE-ABILITY-TO-RESTORE-TEMPORARILY-CHANGED-UNITS
*** When executing SOURCE-DATA-MAINTENANCE, must always be able to restore temporarily changed units so that the actual restoration occurring at the end of the job will not be anomalous.
***

SOURCE-DATA-MAINTENANCE[$] ⇒
admissible RESTORE-UNITS-TEMPORARILY-CHANGED-SINCE[
latest start SOURCE-DATA-MAINTENANCE[$]

II.1.8 *** Maintenance of information about units ***

*** According to the English specification many items of information about units are to be kept within the unit accounting record. Using the specification language we are able to derive such information rather than explicitly assigning it to the units as they are created. To demonstrate how this is possible, we retain two such pieces of information associated with units, *programmer* and *source-language*, and show how these may be derived.
***

```
derivation DERIVE-UNIT-PROGRAMMER(unit)
  whenever ∃ programmer || programmer ≈ ( PROGRAMMER($) as of create unit )
  conclude unit:PROGRAMMER = programmer ;
```

*** PROGRAMMER($) is assumed to be a unary relation provided by the operating environment returning the programmer submitting the current job.
***

```
derivation DERIVE-UNIT-SOURCE-LANGUAGE(unit)
  whenever ∃ language ||
        ( ∃ add-command || add-command:SOURCE-LANGUAGE = language )
        as of latest start ADD[unit,$,add-command]
  conclude unit:SOURCE-LANGUAGE = language ;
```

II.2 HOST-IMP SPECIFICATION

HOST IMP PROTOCOL specified in Gist

Contents:

General features of host-imp world
Nodes
Hardware failure
Specialization to hosts and imps
Interface with external world
Hosts
Imps

```
*** ====================== General features ======================== ***
```

*** Objects of type item are transmitted between nodes. Relation CONNECTED indicates which nodes are connected for transmission.
***

```
type
   item(MESSAGE, ADDRESS | node);

relation    CONNECTED(NODE, NODE);

always prohibit SELF-CONNECTEDNESS
   ∃ node || CONNECTED(node,node);

          *** ------------------- NODES ------------------- ***
```

*** Nodes are agents which perform transmissions. Hosts and imps will be specializations of these. Nodes have several groups of items or messages associated with them:
PENDING - items yet to be dealt with by the node
TO-BE-SENT - items to be sent elsewhere
MY-MESSAGES - messages from items addressed to the node
***

```
agent node( PENDING :any | item ,
            TO-BE-SENT :any | item , MY-MESSAGES :any | message) with

   action
      TRANSMIT[item, to | node]
         precondition   CONNECTED(*self*,to)
                           *** Within context of definition of agent,
                               *self* used to refer to that agent. ***
      *** Invoking TRANSMIT serves as an observable event,
          so this action needs no body ***;


   pending-demon  RECEIVE(item, event)
      *** Watch for transmissions to this node. ***
      trigger  started-event(event)  ∧  event = TRANSMIT(item,*self*)
      response
         begin
            wait until ( finished-event(event) ∨ aborted-event(event) );
            if  finished-event(event)    *** i.e. transmitter remained OK ***
            then  insert item into *self*:PENDING
         end;

   pending-demon  PROCESS-PENDING(item)
      trigger  item = *self*:PENDING
      response
         begin
            delete  *self*:PENDING = item;
            if item:ADDRESS = *self*
            then insert *self*:MY-MESSAGES = item:MESSAGE;
            else insert *self*:TO-BE-SENT = item
         end;


   action
      ACTIVATE[] *** This action is invoked when the agent is activated ***
      definition
         begin
            delete all *self*:TO-BE-SENT = $;
            delete all *self*:PENDING = $;
            wait Relay-Set-Time;
         end

end agent
```

```
*** ---------------------- Hardware Failure ------------------- ***

agent hardware failure with

    *** Node hardware failure is modelled by the following demons, which at random deactivate
    and reactivate nodes.
    ***

  demon NODE-HARDWARE-FAILURE
    trigger random
    response deactivate(some node );

  demon NODE-HARDWARE-RECOVERY(node)
    trigger random ∧ ¬active(node)
    response activate(node)

end agent;

*** ================  Specialization to host-imp world ================ ***

    *** Hosts and imps are specializations of nodes. Messages specialize to user-messages,
    acknowledgments (of user messages, when they reach the imp connected to the destination
    host), and host-going-down-messages (which a host sends to its imp to indicate it is being turned
    off).
    ***

agent node superagentof < host , imp > end agent

type message supertypeof < user-message ;
                           acknowledgments(ACKNOWLEDGES | item) ;
                           host-dead-acknowledgments(ACKNOWLEDGES | item) ;
                           host-going-down-message > ;

always prohibit DIRECT-HOST-CONNECTIONS
  ∃ host.1, host.2 || CONNECTED(host.1,host.2) ;

          *** ------------ INTERFACE ---------------- ***

    *** We model the user interface and possible interactions on our host-imp world by defining
    two demons, USER-SEND to create and send a user-message to some host, and USER-TURN-
    OFF-HOST to turn off a host.
    ***
```

```
agent interface with

   demon USER-SEND
      trigger random
      response
         begin
            create user-message, item ||
                              item:MESSAGE = user-message,
                              item:ADDRESS = some host;
            insert (some host):TO-BE-SENT = item
         end;


   demon USER-TURN-OFF-HOST
      trigger random
      response
         begin
            require ∃ host;
            create host-going-down-message, item ||
                              item:MESSAGE = host-going-down-message,
                              item:ADDRESS = host:CONNECTED-IMP;
            insert host:TO-BE-SENT = item
         end


end agent

            *** ----------------- HOSTS ----------------- ***


      *** Hosts, being subagents of nodes, inherit nodes definitions; in addition they have the
    following:
    ***


agent host(CONNECTED-IMP | imp) with

   always require  ∀ host, imp ||
                   host:CONNECTED-IMP=imp ¿⇒ CONNECTED(host,imp)
                                          ¿⇒ CONNECTED(imp,host)


   pending-demon SEND(item)
      trigger  item = *self*:TO-BE-SENT
      response  TRANSMIT[item , *self*:CONNECTED-IMP]

end agent
```

```
*** ---------------------- IMPS ---------------------- ***

    *** Imps, being subagents of nodes, inherit nodes definitions; in addition they have the
    following:
    ***


agent imp(DEAD-HOSTS :any | host) with

    *** Attribute :DEAD-HOSTS used to record which of the connected hosts are considered to be
    "dead" by the imp.
    ***


    *** One event which causes an imp to mark a connected host as "dead" is receipt of a host-
    going-down-message. ***

  demon  MARK-HOST-DEAD(host, host-going-down-message)
    trigger  host:CONNECTED-IMP = *self*  ∧
             ( host-going-down-message::MESSAGE = *self*:PENDING  ∧
               creator-of host-going-down-message = host ) ∨
             ¬active(host)  )
    response  insert *self*:DEAD-HOSTS = host;


    *** Successful receipt of a message from a connected host (other than a host-going-down-
    message) will cause the host to be removed from those marked as dead. ***

  pending-demon  UNMARK-HOST-DEAD(host, item)
    trigger ¬(item:MESSAGE isa host-going-down-message) ∧ item = *self*:PENDING  ∧
            host = agent-of( latest TRANSMIT(item , *self*) )
    response  delete *self*:DEAD-HOSTS = host;

  pending-demon  SEND(item)
    trigge   item = *self*:TO-BE-SENT
    response
      begin
        delete *self*:TO-BE-SENT = item;
        if CONNECTED(*self* , item:ADDRESS)
        then if item:ADDRESS isa host
             then if item:ADDRESS ¬= :DEAD-HOSTS
                  then begin
                         TRANSMIT[item,item:ADDRESS];
                         if item:MESSAGE isa user-message
                         then ACKNOWLEDGE-OK[item]
                       end
                  else if item:MESSAGE isa user-message
                       then ACKNOWLEDGE-DEAD[item];
             else TRANSMIT[item,item:ADDRESS]

        else TRANSMIT[item,$]
               *** Send item onwards through network. ***
```

```
pending-demon  RECEIVE(item, event)

    *** Specialize definition of receive for imps to terminate wait for a message if more than 15
  seconds have passed since start of receipt. ***

   trigger  started-event(event)  ∧  event = TRANSMIT(item,node)
   response
      begin
         wait until ( finished-event(event) ∨ aborted-event(event)  ∨
                                    duration(event) > 15 seconds );
         if finished-event(event)  then  insert *self*:PENDING = item
      end;


pending-demon  PROCESS-MY-MESSAGES(message)
   trigger  message = *self*:MY-MESSAGES
   response  delete *self*:MY-MESSAGES = message;
          *** Imps simply discard messages addressed to themselves. ***


action ACKNOWLEDGE-OK[item]
   definition
      begin
         create  acknowledgments, reply | item
                              || acknowledgments:ACKNOWLEDGES = item,
                                 reply:MESSAGE = acknowledgments,
                                 reply:ADDRESS = creator-of(item);
         insert reply into *self*:TO-BE-SENT
      end;


action
   ACKNOWLEDGE-DEAD[item]
   definition
      begin
         create  host-dead-acknowledgments, reply
                              || host-dead-acknowledgments:ACKNOWLEDGES = item,
                                 reply:MESSAGE = host-dead-acknowledgments,
                                 reply:ADDRESS = creator-of(item);
         insert *self*:TO-BE-SENT = reply
      end;


demon CONNECTED-HOST-GONE-DOWN(host)
                  *** What to do when a connected host goes down. ***
   trigger  host:CONNECTED-IMP = *self* ∧ ¬active(host)
   response  DEADIFY[host]
end demon
```

```
   action
      DEADIFY[host]
      definition
         begin
            delete all *self*:TO-BE-SENT =( some item || item:ADDRESS = host );
            unschedule all  [ SEND(item) || item:ADDRESS = host ]
         end;


   demon  TARDY-SEND-QUEUES(item,host)
               *** What to do if send queue not emptied rapidly enough. ***
      trigger  item:ADDRESS = host  ∧  host:CONNECTED-IMP = *self*  ∧
                     duration( item = *self*:TO-BE-SENT ) > 30 seconds
      response  DEADIFY[host];
end agent
```

II.3 FORMATTER SPECIFICATION

```
begin
   comment     TYPES       end comment;

   type mixlet() supertype of < info←line(); paragraph(); padding←line() > ;
   comment  Input to our formatter is a sequence of mixlets.  end comment;

   type info←line(il←line | line ::unique , il←info | information :: unique);
   comment  info←lines contain verbatim text, plus layout information.   end comment;

   type paragraph(p←words | sequence of info←word);
   comment  paragraphs contain info←words to be accumulated into justified lines.  end comment;

   type padding←line(p←info | information);
   comment  These are to emerge as blank lines in the output, unless
            they would occur at the very top of a page, in which case
            they are to be discarded.
   end comment;


   type line(chars | sequence of char);

   type char(character);
   type underlined←char() subtype of < char() > ;
   comment  We assume that all chars to be underlined are of this subtype.
            Only the interfaces need do any work - the input to classify
            the appropriate characters to be underlined, and the output
            to produce something appropriate to the physical device
            doing the printing (e.g. backspace and underline character).
   end comment;


   type info←word(iw←chars | sequence of char, iw←info | information ::unique);
   comment  info←words contain a word (some sequence of chars) together with
            layout information.   They occur in paragraphs, and are to be
            accumulated into justified lines.
   end comment;


   type information(header←title | sequence of char,
                    footer←title | sequence of char,
                    left←margin | nat←num,
                    subsequent←left←margin | nat←num,
                    right←margin | nat←num,
                    page←length | pos←int,
                    line←spacing | nat←num,
                    to←begin←page :optional,
                    width | pos←int,
```

```
                    subsequent←width | pos←int)

            bound equiv by = :header←title,
                           = :footer←title,
                           = :left←margin,
                           = :subsequent←left←margin,
                           = :right←margin,
                           = :page←length,
                           = :line←spacing,
                           = :to←begin←page,
                           = :width,
                           = :subsequent←width
            definition
               information:width = information:right←margin - information:left←margin and
               information:subsequent←width = information:right←margin - information:subsequent←width;

   type to←begin←page() supertype of < an←increment(); an←absolute() > ;
   type an←increment(increment←value | integer);
   type an←absolute(absolute←value | integer);

   always prohibited INNER←PARAGRAPH←WORD←TO←BEGIN←PAGE
      there exists info←word
        || info←word:to←begin←page = $ and member(paragraph,(some n | integer || n > 1),info←word);
   comment   info←words of a paragraph beyond the first word may not cause a begin page.
   end comment;

   type nat←num() supertype of < integer() >  where always required ( nat←num > -1 ) end;
   type pos←int() supertype of < integer() >  where always required ( pos←int > 0 ) end

end..
```

```
begin
   action  FORMAT[mixlets | sequence of mixlet]
    definition
      begin
         create mixlets←input || mixlets←input:mixture = mixlets;
         comment   This will trigger demons to do centering, paragraphing,
                   inter-line-padding, splitting of overlength lines,
                   converting space←downs to padding and converting left
                   margin values to leading blanks.   As a result, only
                   text←lines and padding←lines remain in mixlets←input,
                   so the next step is to consolidate these into a simpler
                   data type:
         end comment;
         CREATE←LINELETS←FROM←MIXLETS[mixlets←input:mixture];

         comment   Now do page formation, which will cause a sequence of sequence
                   of lines (representing pages) to be inserted into
                   relation PAGE←OUTPUT.
         end comment;
         PAGINATION[ CREATED←LINELETS←FROM←MIXLETS(*,mixlets←input:mixture) ]
      end;


   type mixlets←input(mixture | sequence of mixlet);

   type linelet(l←info | information :: unique, l←line | (line union padding));
   type padding() definition { Padding };

   relation CREATED←LINELETS←FROM←MIXLETS(linelets | sequence of linelet,
                                          mixlets | sequence of mixlet)
end..
```

```
begin
   action  CREATE←LINELETS←FROM←MIXLETS[ mixlets | sequence of mixlet ]
    definition
      begin
         loop mixlets named mixlet
         do case mixlet of
            info←line
               => begin
                     create linelet || linelet:l←info = mixlet:il←info and
                                       linelet:l←line = mixlet:il←line;
                     insert LINELET←FROM←MIXLET(linelet,mixlet)
                  end;
            padding←line
               => begin
                     create linelet || linelet:l←info = mixlet:p←info and
                                       linelet:l←line = Padding;
                     insert LINELET←FROM←MIXLET(linelet,mixlet)
                  end
          end case;

        insert CREATED←LINELETS←FROM←MIXLETS( mixlets named mixlet obtain
                                                 LINELET←FROM←MIXLET(*,mixlet) ,
                                              mixlets )
       end;

   relation  LINELET←FROM←MIXLET(linelet,mixlet)

end..
```

```
begin
   action  PAGINATION[ linelets | sequence of linelet ]
   definition
      begin
         continuations of
            begin
               comment  Find some arbitrary partition of linelets end comment;
               require there exists page←partition | sequence of sequence of linelet
                        || PAGE←PARTITION(page←partition,linelets);

               comment  Form the page←image (sequence of line) for each sequence
                        of linelets in the partition.
               end comment;
               loop page←partition named page←linelets
                do CREATE←PAGE←IMAGE[page←linelets];

               require there exists page←images | sequence of sequence of line
                        || page←images = ( page←partition named page←linelets obtain CREATED←PAGE←IMAGE(*,
page←linelets) )
            end
         from which page←images satisfies possibility minimizes alternatives wrt length;
         comment  Do the above in such a way as to minimise the number of pages
         end comment;

         insert PAGE←OUTPUT(page←images)
      end
end..
```

```
begin
   relation PAGE←PARTITION( page←partition | sequence of sequence of linelet,
                            linelets | sequence of linelet )
   definition
      ( flatten(page←partition) = linelets )  and
      ( all page←linelets | sequence of linelet
            || member(page←partition,$,page←linelets)
                =>  not member(page←linelets, (some n | integer || n > 1),*):p←info:to←begin←page = $) ;

      comment  page←partition is OK iff it is a partition of linelets, and
               all linelets to begin pages occur as the first linelet of
               their sequence.
      end comment

end..
```

```
begin
   action CREATE←PAGE←IMAGE[ page←linelets | sequence of linelet,
                             page←partition | sequence of sequence of linelet ]
   let page←information | information = first(page←linelets):p←info
   definition
      begin
         require there exists page←number | integer
                  || PAGE←NUMBER(page←number, page←linelets, page←partition);


         CREATE←TITLE[ page←number, page←information:header←title ];
         require there exists heading | sequence of line || heading = CREATED←TITLE(*);

         CREATE←TITLE[ page←number, page←information:footer←title ];
         require there exists footing | sequence of line || footing = CREATED←TITLE(*);

         require there exists center | sequence of (line union padding)
                  || center =
                     STRIP←LEADING←AND←TRAILING←PADDING(*,page←linelets named page←linelet obtain page←lin
elet:l←line) @
                     ( some paddings | sequence of padding);

         require length(heading @ center @ footing) = page←information:page←length;
         insert CREATED←PAGE←IMAGE( heading @ center @ footing , page←linelets )
      end;


   relation  CREATED←PAGE←IMAGE( lines | sequence of line, page←linelets | sequence of linelet ) ;

   relation  CREATED←TITLE( lines | sequence of line )

end..
```

```
begin
   relation PAGE←NUMBER( page←number | integer,
                         page←linelets | sequence of linelet,
                         page←partition | sequence of sequence of linelet )
   definition
      begin
         if there exists to←begin←page || first(page←linelets):l←info:to←begin←page = to←begin←page
         then if to←begin←page isa an←absolute
              then page←number = to←begin←page:absolute←value
              else page←number = PRECEDING←PAGE←NUMBER(*,page←linelets,page←partition)
                                 + to←begin←page:increment←value
         else page←number = 1 + PRECEDING←PAGE←NUMBER(*,page←linelets,page←partition);
         comment  If the first linelet of the page has an absolute to←begin←page value,
                  then that is the number of the page; if it has an incremental
                  to←begin←page value, the number is that increment of the preceding
                  page number; otherwise, simply 1 + the preceding page number.
         end comment
      end;


   relation PRECEDING←PAGE←NUMBER( preceding←page←number | integer,
                                   page←linelets | sequence of linelet,
                                   page←partition | sequence of sequence of linelet )
   definition
      begin
         if there exists preceding←page←linelets | sequence of linelet
              || ( page←partition = $  @  < preceding←page←linelets >  @  $ )
         then PAGE←NUMBER(*,preceding←page←linelets,page←partition)
         else 0;

         comment   Thus the default page number for the very first
                   page (which will have no preceding page) is 1.
         end comment
      end
end..
```

```
begin
   relation STRIP←LEADING←AND←TRAILING←PADDING( core←lines | sequence of (line union padding),
                                                lines | sequence of (line union padding) )
   definition
      ( lines = ( some pre←padding | sequence of padding) @
                core←lines @
                 ( some post←paddings | sequence of padding)  )  and
      not first(lines) isa padding  and
      not last(lines) isa padding

end..
```

```
begin
   relation LINES←FROM←LINELETS( lines | sequence of line,
                                 linelets | sequence of (line union padding) )
   definition
      lines = ( linelets named linelet
                obtain (case linelet of
                          padding => < > ;
                          line    => linelet:1←line
                        end case) )

end.
```

```
begin
   action CREATE←TITLE←LINES[ page←number | integer, title←line | line ]
   definition
      begin
         CREATE←CHARACTERS←FOR←NUMBER[ page←number ];
         require there exists title←lines | sequence of line ||
            ( flatten(title←lines) = flatten( title←line named char obtain
                                              (if PAGE←NUMBER←CHARACTER(char)
                                               then CREATED←CHARACTERS←FOR←NUMBER(*,page←number)
                                               else < char > ) ) )   and
            ( not length(last(title←lines)) > Page←Width  ) and
            ( for all line || ( ( MEMBER(title←lines,$,line) and not line = last(title←lines) ) =>
                               length(MEMBER(title←lines,n,*)) = Page←Width ) ) ;
         update lines in CREATED←TITLE($) to title←lines
      end;

   relation CREATED←CHARACTERS←FOR←NUMBER(characters | sequence of char, number | integer)

end..
```

```
begin
  demon PARAGRAPHING(mixlets←input)
     trigger there exists paragraph, mixlets.1 | sequence of mixlet, mixlets.2 | sequence of mixlet
               || mixlets←input:mixture = mixlets.1 @ < paragraph > @ mixlets.2
     response begin
              comment   Convert the info←words of the paragraph into a
                        sequence of (justified) info←lines.
              end comment;
              CREATE←JUSTIFIED←INFO←LINES←FOR←PARAGRAPH[paragraph];

              update :mixture of mixlets←input
               to mixlets.1 @
                   CREATED←JUSTIFIED←INFO←LINES←FOR←PARAGRAPH(*,paragraph) @
                    mixlets.2
              end;


  relation   CREATED←JUSTIFIED←INFO←LINES←FOR←PARAGRAPH(justified←lines | sequence of info←line,
                                                        paragraph)
end..
```

```
begin
  action
    CREATE←JUSTIFIED←INFO←LINES←FOR←PARAGRAPH[paragraph]
    definition
       begin
          continuations of
             begin
                comment    Pick an arbitrary partition of the paragraph's info←words
                end comment;
                require there exists words←partition | sequence of sequence of info←word ||
                                    flatten(words←partition) = paragraph:p←words ;

                comment   For each sequence of info←words except the last, form a justified info←line
                end comment;
                if there exists all←but←last←words←partition | sequence of sequence of word
                     || all←but←last←words←partition @ < $ > = words←partition
                then begin
                      loop all←but←last←words←partition  named words←to←go←into←one←line
                       do  CREATE←JUSTIFIED←INFO←LINE←FOR←WORDS[words←to←go←into←one←line];

                      comment   Simply fill the last line without justification
                      end comment;
                      CREATE←UNJUSTIFIED←INFO←LINE←FOR←WORDS[ last(words←partition) ]
                     end;

                require there exists justified←lines : sequence of info←line
                          || justified←lines = ( words←partition named words←to←go←into←one←line
                                                 obtain CREATED←INFO←LINE←FOR←WORDS(*,words←to←go←into←one←
line))
             end

          from which justified←lines satisfies possibility minimizes alternatives wrt length;
          comment   Do the above in such'a way as to minimize the number of
                    info←lines produced.
          end comment;

         insert CREATED←JUSTIFIED←INFO←LINES←FOR←PARAGRAPH(justified←lines,paragraph)
       end;



  relation
    CREATED←INFO←LINE←FOR←WORDS(info←line, words | sequence of info←word)
end..
```

```
begin
   action CREATE←JUSTIFIED←INFO←LINE←FOR←WORDS[words | sequence of info←word]
   definition
     if length(words) = 1
      then begin
            comment   Only one word - simply stick it into an info←line.
                      Should this give rise to an overlength info←line, the
                      appropriate demon will be triggered (when the info←line
                      is inserted back into the mixlets←input) to split the line.
            end comment;
            create info←line || info←line:il←line = first(words):iw←chars and
                                info←line:il←info = first(words):iw←info;
            insert CREATED←INFO←LINE←FOR←WORDS(info←line,words)
           end

      else begin
            CREATE←JUSTIFIED←LINE←FOR←WORDS[words];
            create info←line || info←line:il←line =
                                   CREATED←JUSTIFIED←LINE←FOR←WORDS(*,words) and
                                info←line:il←info = first(words):iw←info;
            insert CREATED←INFO←LINE←FOR←WORDS(info←line,words)
           end;


   relation  CREATED←JUSTIFIED←LINE←FOR←WORDS(line, words | sequence of info←word)
end..
```

```
begin
   action  CREATE←JUSTIFIED←LINE←FOR←WORDS[words | sequence of info←word]
   definition
    begin
      ARBITRARILY←PAD←WORDS←WITH←BLANKS[words];
      require there exists info←line
            || ARBITRARILY←PADDED←INFO←LINE(info←line,words);

      comment  Require that the line be of the right length (i.e. justified)
               - thus our arbitrary padding wasn't quite so arbitrary after all.
      end comment;
      require info←line:il←info:width = length(info←line:il←line);

      insert  CREATED←JUSTIFIED←LINE←FOR←WORDS(info←line,words)
    end;

    relation ARBITRARILY←PADDED←INFO←LINE(info←line, words | sequence of word)
end..
```

```
begin
   action ARBITRARILY←PAD←WORDS←WITH←BLANKS[words | sequence of word]
   definition
    begin
      create word←blank←seq  || word←blank←seq:wbs = words;

      comment  Next do some (arbitrary) padding:  end comment;
      PAD←INTERIOR←WITH←BLANKS[word←blank←seq];

      comment  Require that no unseparated words remain.   end comment;
      require not there exists info←word.1, info←word.2
            || word←blank←seq:wbs = $ @ < info←word.1, info←word.2 > @ $;

      create info←line ||  info←line:il←info = first(words):iw←info and
                           info←line:il←line = flatten(word←blank←seq:wbs named w←or←b
                                                       obtain
                                                        (case w←or←b of
                                                         word => w←or←b:chars;
                                                         char => < w←or←b >
                                                         end case));

      insert  ARBITRARILY←PADDED←INFO←LINE(info←line,words)
    end;

   type word←blank←seq(wbs | sequence of (info←word union blank))
end..
```

```
begin
   action  PAD←INTERIOR←WITH←BLANKS[word←blank←sequence]
   definition
      choose
         begin require there exists wbs.1 | sequence of (word union blank),
                                    wbs.2 | sequence of (word union blank)
               || word←blank←sequence:wbs = wbs.1 @ wbs.2 ;
         create new←blank | char || new←blank:character = Blank;
         update :wbs of word←blank←sequence to
                        wbs.1 @ < new←blank > @ wbs.2;
         PAD←INTERIOR←WITH←BLANKS[word←blank←sequence]
         end;

         comment  Or choose to do no more padding.  end comment
       end choose
end..
```

```
begin
   action CREATE←UNJUSTIFIED←INFO←LINE←FOR←WORDS[ words | sequence of word ]
   definition
      begin
         continuations of
         begin
               ARBITRARILY←PAD←WORDS←WITH←BLANKS[words];
               require there exists info←line
                     || ARBITRARILY←PADDED←INFO←LINE(info←line,words);

               comment  Require that the line not be overlength  end comment;
               require not length(info←line:il←line) > info←line:il←info:width
            end
         from which info←line:il←line satisfies possibility minimizes alternatives wrt length;

         insert CREATED←INFO←LINE←FOR←WORDS(info←line,words)
      end

end..
```

```
begin
   demon  SPLIT←OVERLENGTH←LINES( mixlets←input )
   trigger there exists info←line, mixlets.1 | sequence of mixlet,
                        mixlets.2 | sequence of mixlet
                   || mixlets←input:mixture =
                      mixlets.1 @ < info←line > @ mixlets.2  and
                     OVERLENGTH←INFO←LINE(info←line)
   response
      begin
         comment   Any overlength line is to be split into lines
                   of acceptable length.
         end comment;
         CREATE←SPLIT←LINES←FOR←OVERLENGTH←LINE[ info←line ];

         update :mixture of mixlets←input
         to  mixlets.1 @
              CREATED←SPLIT←LINES←FOR←OVERLENGTH←LINE(*,info←line) @
               mixlets.2
      end;


   relation CREATED←SPLIT←LINES←FOR←OVERLENGTH←LINE( split←lines | sequence of info←line,
                                                     long←line | info←line )
end..
```

```
begin
   action CREATE←SPLIT←LINES←FOR←OVERLENGTH←LINE[ info←line ]
   definition
      begin
         comment   The info←line is overlength, so split it into two info←lines,
                   making sure the first line is of acceptable length (if the
                   remainder is still overlength, then the demon will cause
                   a split of it, and so on).
         require there exists line.1, line.2
                  || info←line:il←line = line.1 @ line.2 and
                     length(line.1) = info←line:il←line:width;
         update :il←line of info←line to line.1;

         comment   The information to go into the second line is "subsequentized",
                   both to ensure that if the original line was to start a
                   page, only the first line formed by the split will do the same,
                   and to adjust the left margin of the remainder.
         end comment;
         atomic
            create information, info←line.2 || information equiv info←line:iw←info and
                                               info←line.2:il←line = line.2 and
                                               info←line.2:il←info = information;
            SUBSEQUENTIZE←INFO[ info←line.2:iw←info ]
         end atomic;

         insert CREATED←SPLIT←LINES←FOR←OVERLENGTH←LINE(<info←line,info←line.1>, info←line)
      end
end..
```

```
begin
   action SUBSEQUENTIZE←INFO[ information ]
   definition
      begin
         update :left←margin of information to information:subsequent←left←margin;
         delete all information:to←begin←page = $
      end


end..
```

```
begin
   demon LEFT←MARGIN←PADDING( mixlets←input )
   trigger there exists info←line, mixlets.1 | sequence of mixlet,
                                    mixlets.2 | sequence of mixlet
            || mixlets←input:mixture = mixlets.1 @ < info←line > @ mixlets.2  and
              info←line:il←info:left←margin > 0  and
              not OVERLENGTH←INFO←LINE(info←line)  and
              not TO←BE←CENTERED(info←line)
   response
      begin
         comment   info←lines with non-zero left←margin values are adjusted
                   to begin with the appropriate number of blanks.  Note that
                   this will not be done before the info←line has been
                   split (if it is overlength) and centered if need be.
         end comment;
         create blank←sequence || blank←sequence:bs = < >;
         EXTEND←BLANK←SEQUENCE[blank←sequence];
         require length(blank←sequence:b←s) = info←line:il←info:left←margin;

         atomic
            update :left←margin of info←line:il←info to 0;
            update :il←line of info←line to blank←sequence:b←s @ info←line:il←line
         end atomic
      end;

   type blank←sequence(b←s | sequence of character)

end..
```

```
begin
   action EXTEND←BLANK←SEQUENCE[ blank←sequence ]
   definition
      choose
         begin
            create char || char:character = Blank;
            update :b←s of blank←sequence to <char> @ blank←sequence:b←s;
            EXTEND←BLANK←SEQUENCE[ blank←sequence ]
         end;
         comment  Or choose not to extend any further.  end comment
      end choose

end..
```

```
begin
   demon  CENTERING( mixlets←input )
      trigger there exists info←line, mixlets.1 | sequence of mixlet,
                                    mixlets.2 | sequence of mixlet
                || mixlets←input:mixture = mixlets.1 @
                    < info←line > @
                     mixlets.2  and
                   TO←BE←CENTERED(info←line)
      response
         begin
            comment   This demon causes adjustments to the left margin value in
                      order to effect centering of text between margins.
            end comment;
            atomic
               delete TO←BE←CENTERED(info←line);
               update :left←margin of info←line:il←info
                to (if not OVERLENGTH←INFO←LINE(info←line)
                    then half(info←line:il←info:width - length(info←line:il←line))
                    else 0)
            end atomic

         end
end..
```

```
begin
   demon INTER←LINE←PADDING( mixlets←input )
   trigger there exists mixlets.1 | sequence of mixlet,
                        mixlets.2 | sequence of mixlet,
                        info←line
            || mixlets←input:mixlets = mixlets.1 @ < info←line > @ mixlets.2  and
               info←line:il←info:line←spacing > 0 and
               not OVERLENGTH←INFO←LINE(info←line)
   response
      begin
         comment   This demon creates padding←lines to do the spacing
                   between text lines.  Note that padding is not to be
                   added after an overlength line - this is because such
                   a line will be split into several lines by a separate
                   demon, after which padding may be performed safely.
         end comment;

         require there exists no←of←padding←lines | integer
                   || no←of←padding←lines = info←line:il←info:line←spacing;
         update :line←spacing of info←line:il←info to 0;

         comment   Take care to ensure the new padding←lines inherit
                   appropriately "subsequentized" information - we
                   don't want them starting a new page simple because
                   the line they are following is to do so.
         end comment;
         atomic
            create new←info | information || new←info equiv info←line:il←info;
            SUBSEQUENTIZE←INFO[ new←info ]
         end atomic;

         create padding←line←sequence || padding←line←sequence:p←l←s = < >;
         EXTEND←PADDING←LINE←SEQUENCE[ padding←line←sequence, new←info ];
         require length(padding←line←sequence:p←l←s) = no←of←padding←lines;

         update :mixture of mixlets←input
          to mixlets.1 @ < info←line > @ padding←line←sequence:p←l←s @ mixlets.2
      end

end..
```

```
begin
   action EXTEND←PADDING←LINE←SEQUENCE[ padding←line←sequence | sequence of padding←line,
                                        info←to←copy | information ]
   definition
      choose
         begin
            create information, padding←line || information equiv info←to←copy and
                                                padding←line:p←info = information;
            update :p←l←s of padding←line←sequence
             to < padding←line > @ padding←line←sequence:p←l←s;
            EXTEND←PADDING←LINE←SEQUENCE[ padding←line←sequence, info←to←copy ]
         end;
         comment   Or choose not to extend any further.   end comment
      end choose

end..
```

REFERENCES

1. Balzer, R., *Transformational Implementation: An Example*, USC/Information Sciences Institute, RR-79-79, 1979.

2. Kernighan, B. W., and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

3. Teitelman, W., *Interlisp Reference Manual*, Xerox Palo Alto Research Center, October 1978.

4. Tinanoff, N., and F. M. Luppino, *Programming Support Library (PSL) Program Specifications*, , Technical Report RADC-TR-74-300-vol-6, November 1974. (Structured Programming Series, Volume VI, AD-A007 796.)

5. Wile, D. S., POPART: Producer of parsers and related tools, 1981. (in preparation).

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements.. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.